SAND2017-0145R

IDC RE-ENGINEERING REPORT

SAND2016-WXYZ
Unlimited Release
December 2016

IDC Re-Engineering Phase 2 Architecture Document

Version 1.7

John Burns

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

SAND2016-WXYZ
Unlimited Release
December, 2016

IDC Re-Engineering Phase 2 Architecture Document

J. Burns
Next Generation Monitoring Systems
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0401

Abstract

This document contains a description of the system architecture for the IDC Re-Engineering Phase 2 project.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 The US NDC System

The Air Force Technical Applications Center (AFTAC) operates the US National Data Center (US NDC) to support the US government nuclear test treaty monitoring mission. The US NDC system integrates, processes, and analyzes data in order to detect, locate, identify, and report nuclear events. It also collects and forwards unprocessed data to US researchers.

1.2 US NDC Modernization Project

The current US NDC system meets mission requirements, but is difficult and expensive to maintain and enhance. The system and its uses have evolved considerably since its initial design. The U.S. government has initiated a project to modernize the current system - the US NDC Modernization project - in order to address these shortcomings, and to enable continued mission support into the future. The primary purpose of this project is to update the system architecture and software, applying modern processes, software design principles and technologies. Sandia National Laboratories (SNL) has been contracted to execute initial work on the project.

1.3 The IDC System

The International Data Centre (IDC) of the Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) applies, on a routine basis, automatic processing methods and interactive analysis to raw International Monitoring System (IMS) seismic, hydroacoustic, infrasonic, and radionuclide data in order to produce, archive, and distribute standard IDC products on behalf of all States Parties. The routine processing includes analysis of events with the objective of screening out events considered to be consistent with natural phenomena or non-nuclear, man-made phenomena.

1.4 IDC Re-engineering

The current IDC System was designed in the early 1990s, and needs to be redesigned to be consistent with modern practices. The goal of the IDC Re-engineering Phase 2 project is to develop an architecture for a modernized version of the IDC System. The scope of the project is limited to seismic, hydroacoustic, and infrasonic except for product distribution, which applies to all four technologies.

1.5 The Architecture Document

The purpose of this document is to describe the analysis and design methodology, overall software structure, common architectural patterns, core system mechanisms, and design rationale for a common system to support the IDC Re-engineering Project and the US NDC Modernization Project. In the remainder of the document the term "System" refers to this common system. The Architecture Document is one of the primary artifacts created and maintained by the architects; it serves as a communication medium between the architecture team and the implementation teams.

2 DOCUMENTS

Table 2-1. Documents

Author/Document No.	Title	Rev./Date
CTBTO	IDC Re-Engineering Phase 2 System Requirements Document, V1.4	Jan 2017
IDC-RP2-SSD-V1.5	IDC Re-Engineering Phase 2 System Specification Document	Jan 2017
—END OF TABLE—		

3 SCOPE

The Architecture Document captures the general principles used throughout the design of the system and serves as a guide for the further development of the system. It is not meant to be a complete design document. To keep it short and accessible, and to avoid too much redundancy with other documents, this Architecture Document is intentionally limited to a high level of abstraction.

The Architecture Document evolves during the design of the system and continuously captures new major design decisions made regarding the system. In the context of an iterative development process, there is one version of the Architecture Document per development iteration.

The terminology and graphical notation used in this document are derived from the Unified Modeling Language (UML).

3.1 Mission

The mission of the System is to monitor compliance with nuclear test treaties. This mission requires the System to detect, locate, evaluate, store, and report natural and man-made events. The System uses several different monitoring techniques to perform this mission, each designed to monitor a specific physical domain (e.g., atmosphere, underground, underwater) for events.

3.2 Deployment Concept

The System is designed to be operational 24 hours per day, 7 days per week (24x7). The System can be deployed at multiple, redundant fixed sites as well as limited functionality deployed on stand-alone, portable systems.

3.3 Modernization Principles

Modernization is necessary to maintain current mission capability, reduce the cost of software maintenance, and provide the ability to enhance the system into the future. The following principles support achieving the goals of the modernization program:

- Re-architect System using Model-Based Engineering
- Enhanced Mission Capabilities
- Extensibility
- Scalability
- Platform Independence
- Integrated Testing

- Modernize Development Process and Environment

3.3.1 Re-Architect System Using Model-Based Engineering

Model-based engineering is a methodology that focuses on creating and maintaining a set of models that describe different aspects of the design of the system. During the design and development process the models provide an abstraction to facilitate discussion of significant system features. Models provide a means of capturing design decisions and rationale. Additionally, models provide a basis for on-going support and enhancement of the system. Model-based engineering is a key to most modern software lifecycle processes.

3.3.2 Enhanced Mission Capabilities

One of the primary objectives of this effort is to enhance existing data acquisition, event detection, data distribution, and data retention capabilities to meet current and future treaty monitoring needs as specified in the System Requirements Documents (SRDs). The design of the modernized system integrates improved seismic, hydroacoustic and infrasonic propagation models (velocity, attenuation, etc.) to improve association, location, magnitude estimation and event screening or identification. Another area of focus is exploiting historic data captured by the System to improve automated system performance. The System captures detailed histories of analyst interactions with the system to support refining the system and improving diagnostic capabilities.

3.3.3 Extensibility

The System is designed to support continued expansion and refinement of the mission requirements. The system software is based on open standards and leverages software trends supporting extensibility. The system design facilitates integration of new tools and models to permit continued evolution and improvement of the system. Algorithms and models are key areas of improvement for the system so the design will describe interfaces and loose coupling of components to permit addition or substitution of these components. The system will also allow system maintainers to modify processing sequences and parameters to permit tuning system performance without extensive redesign and recoding. Another area of focus for the design is operator customization. The system will use commercial off-the-shelf (COTS), open-source software and user interface frameworks to permit users to tailor their displays to their work preferences so users can be efficient.

3.3.4 Scalability

The System is scalable to permit growth of the System to support increasing requirements in number of stations, in the amount of historical data supporting

operational processing, in the number or complexity of processing algorithms. Scalability considerations in the architecture facilitate evolution of the System by providing a design that can easily migrate to additional processing capability, increased network throughput or expanded memory. Another aspect of scalability is the incorporation of new algorithms based on "Big Data" analysis methods. "Big Data" is a term for data sets that are so large or complex that traditional data processing applications are inadequate to deal with them. The current architecture concept for the Modernized System is a hybrid solution incorporating both a 'Big Data' platform focused on research support as well as a traditional real-time processing architecture. The current architecture focus is on developing traditional real-time processing architecture. As "Big Data" analysis methods are researched and developed for operations these methods would be implemented in parallel to the real-time processing.

3.3.5 Platform Independence

To facilitate continued system improvement and to avoid vendor-lock the modernized system will emphasize open-source operating systems and standard frameworks.

3.3.6 Integrated Testing

A fundamental aspect of verifying the System will be the ability to reproduce a set of inputs to the system and capture results of processing that data. The System will capture raw input data and provide the capability to replay the data through the System. The System will capture provenance information about system performance, processing parameters and intermediate results to facilitate analysis and debugging.

3.3.7 Modernize Development Process and Environment

A major objective of this effort is to make use of modern software design and development practices to re-architect the System's legacy source code/software baseline that has evolved over the past several years. This effort shall include recoding the software baseline to take advantage of modern object-oriented software languages and 64-bit multi-core central processing unit (CPU) architectures. The architecture design will utilize software modeling tools and processes to communicate, review and maintain the architectural design of the modernized system. Common tool sets at geographically distributed development sites will facilitate a shared understanding and implementation of the architecture.

3.4 Key Features

The System supports these new key features to enhance effectiveness and usability: Common Object Interface (COI), provenance, undo-redo, replay,

algorithm extensibility, remote user interface, and Geographic Information Systems (GIS).

3.4.1 Common Object Interface

The Common Object Interface (COI) is a definition of the data persisted on the system. The definition is distinct from the actual format of the stored data to permit the storage format to be optimized without affecting the application software. Persistence of stored information on the system is described in 4.2.6.3 Object Storage and Distribution Mechanism. The Data Model encompasses the detailed design of the **Entity** classes in the System including class attributes and associations. The COI is the interface specification defining how **Entity** classes are accessed in the System.

3.4.2 Provenance

Provenance, as it relates to the US NDC System, encompasses preserving a complete history of the process of forming and refining events. Provenance includes identifying the sensor data used to create an event, the signal processing operations performed on the input signals, algorithms used to form the event, personnel analyzing the event data and interim event solutions. Provenance facilitates post analysis of each event to tune processing algorithms, to understand analyst decisions, and ultimately to recreate the event analysis results.

The architecture captures provenance information in several ways. Provenance information records when Entity classes are created, for example; when a particular instance of waveform data was available. The provenance of events is captured by creating a series of Event Hypotheses that describe versions of the event as the solution evolves over time. Similarly, versions of signal detections are captured as Signal Detection Hypotheses. Relationships between Entity classes such as associations between Event Hypotheses and Signal Detection Hypotheses are stored with a related time range. This method allows analysis of how Entity classes were related as analysis progressed. Provenance includes capture of parameters used including whether the default settings for a particular calculation were used or whether the values were overridden by an analyst. Provenance includes records of which Entity classes (e.g. Event, Event Hypothesis, Waveform, etc.) exist in the System, when those Entity classes are created, which user or process created those Entity classes, which prior Entity classes were used to derive the new Entity classes, and timing information indicating when Entity classes were created and when processing affecting those Entity classes executed.

The architectural approach to capturing provenance is to develop new provenance Entity classes, associate those provenance objects to the appropriate Entities, and store the provenance information in existing data stores. This

approach groups the provenance information directly with the Entity classes. A limitation of this approach is the provenance information needs to be defined beforehand. If additional data is needed to analyze a problem it is not available until the software is updated since the necessary data was not collected when the problem occurred. Additional provenance information beyond what is represented in the provenance Entity classes will be captured in System logs. Log analysis can then be used, if necessary, to determine in more detail than is available in the provenance Entities why the System behaved in a particular manner, why certain results were created, or why certain results were not created.

3.4.3 Undo-Redo

A required feature of the System is the ability for Analysts to undo or redo commands during an analysis session. The intent of undo and redo is to permit the Analyst to revert back to a previous state to recover from an incorrect selection quickly. The Analyst also has the option of saving multiple different Event Hypotheses for an Event to explore alternate solutions for an Event. Therefore undo-redo is intended to cover reverting back a limited number of analysis steps for a single Event.

The System maintains a buffer of Analyst-entered commands. The Analyst may select to step backward or forward through the list of commands that the Analyst has entered and the System returns the Event currently being analyzed to the state associated with the selected command. This requires the System to undo or redo both the Analyst entered commands as well as any automatic processing initiated by the System in response to the Analyst's commands. Signal detection modifications during an Event's analysis are also undo-able. The context of the undo-redo actions is a single Event. If the Analyst works multiple Events concurrently then an undo-redo stack is created for each Event. The Event's undo-redo stack is cleared when the current Event Hypothesis for an Event is saved. Analysts have the capability to save multiple Event Hypotheses of an Event during an analysis session to capture key points in the analysis. Analysts may open saved copies of Events to review prior states or to continue analysis beginning from the prior saved state. If no Event is being analyzed, for example during a waveform scan; then Signal Detection modifications are saved in a separate stack. If the Analyst is scanning waveforms and Signal Detections the stack is cleared when the scanning activity is completed.

Undo-able actions are defined as actions that cause significant state changes in Event Hypotheses or Signal Detections. Actions such as waveform filtering changes are not inherently undo-able because the filter change does not directly change the Event Hypothesis or Signal Detections and the previous filter state is easily obtained by applying a previous filter. However, any waveform filter changes are undone to a previous state to be consistent with the previous state

of the Event if the corresponding Event is undone. Analyst actions that are not undo-able are not shown on the undo-redo stack.

Undo and redo requires that the intermittent states of Events and Signal Detections are either stored or recoverable. The states are directly associated with the undo-able steps. A direct approach to implement this feature is to transiently save a version of the Event and associated Signal Detections at every undo-able step. To undo or redo an action the System reverts to previous Event and Signal Detection states associated with the step.

3.4.4 Replay

The System requires two different modes of replay. The first mode is the capability to capture waveform data and replay the data through the system to support algorithm development and testing. The second mode is the capability to replay Analyst actions to recreate software issues and to support regression testing. Establishing a baseline configuration is essential for both methods of replay. Replay of waveform data depends on configuring the System under test with the identical station and channel configuration as the configuration on the System where the waveform data was captured. To replay Analyst actions the configuration of the System under test must also match the System where the Analyst actions were captured. Configuration entails many diverse aspects of the system and the method for collecting and transferring this information to the System under test will require significant development effort. The following descriptions assume the required configuration is installed on the system under test.

3.4.4.1 Waveform Replay

Waveform replay reproduces the time sequence of data arrival at the system. Waveform replay supports re-creation of error conditions, testing error fixes or testing of new capabilities under known conditions. Waveform replay can recreate the situation where incomplete or corrupted data segments are corrected by later arriving data segments. Waveform replay also enables examination of automatic processing of late arriving data after events have been analyzed.

Waveform replay can be implemented by injecting that data into the system through a test injection interface to emulate the external interfaces. A copy of all data received is re-sent to the system at the appropriate time. This approach requires saving the input data stream, copying the stream to the test injection interface, and sending the input stream to the remainder of the system.

An alternate approach records the reception time for each segment of data as the segments are stored. Then the test data injection process uses the reception times to forward data to the remainder of the system at the appropriate time to

emulate processing the data as it was originally received. This approach requires the time on the system under test to be consistent with the time of the waveform data. An advantage of this approach is the waveform replay does not rely on alternate external interfaces and all received data will potentially be available for replay without transferring the data to the test data injection process. However, because the test replay is embedded in the system, separation of test and operational data will need to be enforced procedurally and/or through separation of test and operational results.

The most critical timing sequence to emulate for evaluating system performance is the processing sequence of waveform data when it arrives at the Data Processing partition. Therefore, waveform replay will be designed to inject data to the Data Processing partition at the time and time sequence of the original data. Additionally, the System will provide the capability to simulate operator load on the system while the waveform data is being replayed.

There are some inherent limitations in reproducing an identical environment during testing even with the replay capability. Small timing differences of data availability during testing because of varied system loads or test platform configuration may affect the test results. Also, it is difficult to reproduce the entire state of the system, especially where algorithms use historic data in analyzing current inputs. In spite of these limitations, the waveform replay capability will greatly expand the ability to test the system.

3.4.4.2 Analyst Action Replay

Analyst action replay allows replication of System response to user interaction with the system to allow more detailed error analysis. The replay capability requires that the set of Analyst actions for one Analyst has been captured and transferred to the System under test. The set of data under analysis must also be replicated to the test System with identical data element identifiers. The data element identifiers must match to ensure that replayed Analyst actions are executed on the same waveforms, Signal Detections and Events. Analyst actions can be replayed at the recorded rate or at an accelerated rate. If accelerating the replay the system clocks must be synchronized to the accelerated rate. To evaluate the result of the Analyst action replay the tester will need to compare the resulting set of Events and Signal Detections with the previous or expected values.

3.4.5 Algorithm Extensibility

Extensibility of key algorithms is key for maintaining and improving the System. Algorithm extensibility allows the algorithms to be updated or replaced without affecting the remainder of the applications. Algorithm extensibility is modeled using **Plugin** classes as described in 4.2.3 Types of Analysis Classes and 4.2.7.4 Algorithms for Automatic and Interactive Processing.

3.4.6 Remote User Interface

An important consideration for the System is to provide a responsive user interface when the analyst is accessing the data remotely. The design of the user interface data communication needs to be optimized to minimize the delays introduced when the data is accessed remotely.

3.4.7 Geographic Information Systems (GIS)

Maps and geospatial processing are key components of system. A GIS can provide much of the needed capabilities but the GIS must be integrated with the remainder of the system. To facilitate integration of alternate GIS implementations GIS functionality should be accessed through a standard interface protocols defined by the Open Geospatial Consortium (OGC). Various standards which might be applicable include Web Map Service (WMS), Web Feature Service (WFS), Web Coverage Service (WCS).

3.5 Assumptions

The following assumptions were used in developing the architecture:

- The sensor network will be similar in type, size, and complexity as currently defined with the capability to expand as required.
- A similar number of operators, analysts, and evaluators that support the current System will support the modernized System.
- A redundant, duplicate processing System at an alternate location will support mission continuity and disaster recovery.
- An iterative design approach will be used for the complete lifecycle of the project.
- The project will be a collaborative effort between SNL and the operating agencies. The operating agencies have allocated significant staff resources to support this effort.
- The mission must execute fully and efficiently during transition to the modernized System.
- The System must be extensible to meet needs for the next ~20 years.

4 ARCHITECTURAL REPRESENTATION

Architecture is a concept that is easy to understand, but is hard to define precisely. In particular, it is difficult to draw a sharp line between design and architecture—architecture is one aspect of design that concentrates on some specific features.

The System is inherently complex. To reduce the complexity, we decompose the system into smaller components or *objects* through the use of Object-Oriented Design methodology. Objects abstract from the complexity by hiding the unimportant details and focusing on the important characteristics and operations.

The decomposition, abstraction and hierarchy can be developed along several orthogonal views, depending on the usage of the system:

- From the ground system application point of view, entities of the real world are mapped onto corresponding design entities; for example, abstractions such as Sensors, Stations, etc.
- From the end-user's (operator's) point of view, certain inputs trigger actions to take place concurrently or sequentially throughout the system.
- From the system designer or programmer point of view, the system is organized in a way that helps its construction, that facilitates its development by several teams concurrently, that helps the long-term management of the software and hardware, and that facilitates the reuse of components throughout the system.
- From the operational and testing points of view, the system can be fully exercised according to the use cases and scenarios. A use case is a sequence of actions performed by the system that yields an observable result.

The system architecture attempts to capture and describe all these aspects rigorously and systematically. It offers multiple views or models of the software, each developed according to its own well-defined set of rules, each revealing one aspect of the system, but all consistent with one another.

In the Rational Unified Process (RUP) used to develop the System, analysis starts from a typical set of views, called the *4+1 View Model*. It is composed of:

- **Use Case View** – Contains the use cases that encompass architecturally significant behavior, classes or technical risks. A use case defines the functions of the system by describing actor interaction with the system.
- **Logical View** – Realizes each use case through the use of high-level Analysis Classes depicted on Unified Modeling Language (UML) class and sequence diagrams. Analysis Classes describe a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.
- **Implementation View** – Organizes Analysis Classes into modules. The Implementation View addresses the organization of delivered source code modules in order to facilitate software development, manage subsystem reuse and reduce subsystem dependencies. Modules are then mapped into layers; yet another level of organization intended to reduce the overall complexity of the software.

- **Process View** – Contains the description of the processes in the system, their interactions and configurations, and the allocation of Analysis Classes to processes. This view is needed because the system has a significant degree of concurrency.
- **Deployment View** – Contains the description of the various physical processing platform configurations, and the allocation of processes (from the Process View) to the physical platforms. This view is needed because the system is distributed.

In addition to the above views, the term *architecture* also includes the overarching patterns and/or frameworks that serve to shape the software.

4.1 Use Case View

Use cases (UCs) provide a basis for describing the system from a user-level or external perspective. Because use cases describe the functionality of the system from the user's point of view, additional context or requirements are discovered from developing and reviewing the use cases. In turn, use cases provide a basis for organizing and analyzing the Logical View and Process View.

In a complex system there are many use cases, some more complicated than others. An important job of the Architecture team is to determine up front which use cases are likely to have significant impacts on the final architecture of the system. Such use cases are referred to as *architecturally-significant* and should be architected into the system first to stabilize the architecture for later work. The determination as to whether a given use case is architecturally-significant or not is largely a subjective matter, but in general is based on answers to the following questions:

- Will the use case have a strong influence on how the overall system is architected; for example, will it require a certain framework to be put in place, or certain widespread assumptions to be made?
- Will implementation of the use case involve many architectural elements (many interfaces, processes, displays, etc.)?
- Does the use case represent an important (perhaps mission-critical) interaction with the system?
- Will implementation of the use case involve higher-than-average technical risk (e.g., excessive data rate or data storage, interfacing with a new external system for the first time)?

The following list shows the use cases defined as *architecturally-significant* for the System:

- 1.1 **System Receives Station Data** – This use case is architecturally significant because it describes acquiring data from multiple sources in various formats and protocols and distributing the data to processing components within timeliness requirements.
- 1.3 **System Acquires Meteorological Data** – This use case is architecturally significant because it describes acquiring meteorological data from various sources and creating a dynamic atmospheric transmission model to support the prediction infrasonic signal detections.
- 2.3 **System Detects Events using Waveform Correlation** – This use case is architecturally significant because waveform correlation requires high levels of processing and memory resources for evaluating large sets of historic data for relevance to real-time events.
- 2.6 **System Builds Events using Signal Detections** –This use case is architecturally significant because it involves complex algorithms for automatically building and modifying events.
- 2.12 **System Predicts Signal Features** – This use case is architecturally significant because it involves use of large and complex earth models for calculation of signal propagation through the earth, including time-varying models of the atmosphere and ocean.
- 3.2 **Refines Event** – This use case is architecturally significant because it encompasses interaction between a large number of capabilities available to Analysts, including synchronized interaction among those capabilities, the Analyst ability to initiate automatic processing algorithms with overridden system parameters, and capture and display of provenance for Analyst actions.
- 3.2.8 **Compares Events** – This use case is architecturally significant because it provides a capability to view and compare the analysis and provenance of multiple events.
- 3.3 **Scans Waveforms and Unassociated Detections** – This use case is architecturally significant because it provides a platform for the Analyst to efficiently review large amounts of sensor data in order to evaluate, correct, and improve signal detection and event formation results.
- 3.5 **Marks Processing Stage Complete** – This use case is architecturally significant because it describes how Analysts complete their defined analysis activities in the context of a processing stage in order to transition control to the next processing stage.
- 5.2 **Views System Results** – This use case is architecturally significant because it provides an interactive method for large number of external customers to access a high volume and diverse set of system results in a timely manner.

- 6.3 **Defines Processing Sequence** – This use case is architecturally significant because it drives the system architecture to support flexible and extensible definition of processing and analysis control flow.
- 6.7 **Views System Configuration History** – This use case is architecturally significant because it defines a new capability of the system to store and view the system configuration at any point in time to support analysis of the impact of configuration changes.
- 7.1 **Analyzes Mission Performance** – This use case is architecturally significant because it describes the display and analysis of a rich set of metrics to assess system mission performance and tune the system.
- 8.2 **Controls the System** – This use case is architecturally significant due to the System's timeliness requirements to start and stop the System and to transfer mission assignment from the Primary to the Backup.
- 8.5 **Views Event History** – This UC is architecturally significant because it describes viewing and comparing multiple versions of an event to review the history of how an event was formed and what data was available at each stage of event formation.
- 9.3 **Replays Test Data Set** – This use case is Architecturally Significant because it describes a testing capability to duplicate system configuration and to inject captured raw data into the system to support testing of error fixes or newly developed capability under known conditions.
- 11.2 **Develops New Algorithms and Models** – This use case is architecturally significant because it drives the system architecture to provide interfaces to System data and processing components accessible to Researchers through command line interfaces and a Common Object Interface (COI).
- 13.2 **Performs Standalone Analysis –** This use case is architecturally significant because it drives the System architecture to support configurable software distributions at various scales of data processing, computing hardware, and personnel to support third-party organizations performing similar monitoring functions.

4.2 Logical View

The Logical View depicts Analysis Classes and their behavior in the context of specific use cases. The functionality described in a use case is mapped to Analysis Classes in Use Case Realizations. The following sections describe Analysis Classes identified in the architecture and the frameworks in which they operate.

4.2.1 Domains

The System is composed of the following domains. These domains provide a logical organization for the Analysis Classes of the system:

- **Data Acquisition** – The Data Acquisition Domain contains classes for handling data reception from various data providers, forwarding the raw data, and storing the data for subsequent access by the System. The System acquires data from stations, from external data centers, and from other sources. Data can be acquired in a variety of formats including CD-1.0, CD-1.1, IMS-1.0, SEED, and miniSEED and the system is extensible to acquire data in new formats in the future.
- **Data Quality –** The Data Quality Domain contains classes for detecting errors in incoming waveforms that can lead to problems with processing and analysis. The data containing errors is identified and masked to prevent further processing. System users can override the system's quality determination. Data quality errors include data gaps, amplitude spikes, repeated amplitude values, linear trends and invalid gain.
- **Signal Enhancement –** The Signal Enhancement Domain contains classes for applying signal processing techniques to enhance the signal content and reduce the noise content of waveform data. The techniques include filtering, beamforming, and three component waveform data rotation.
- **Signal Detection –** The Signal Detection Domain contains classes for using various techniques to identify signals of interest. Signal detections are stored and further processed to identify events.
- **Signal Feature Measurement –** The Feature Measurement Domain contains classes for measuring features associated with a signal detection (e.g., arrival time, back azimuth, horizontal slowness, amplitude, frequency content). The feature measurements are used to analyze the signal detection.
- **Signal Detection Association –** The Signal Detection Association Domain contains classes for using observed and predicted signal detection features to associate signal detections with either new events or existing events.
- **Waveform Correlation –** The Waveform Correlation Domain contains classes for finding new events by matching current waveforms to waveforms of known historical events. A matching new event is created at the location of the historical event.
- **Event Conflict Resolution –** The Event Conflict Resolution Domain contains classes for resolving cases where signal detections are assigned to more than one event. Each signal detection should be associated to at most one event.
- **Event Location –** The Event Location Domain contains classes for determining an event's spatial location and temporal location and uncertainties.

- **Event Magnitude –** The Event Magnitude Domain contains classes for estimating the size of an event by combining the available event magnitudes computed from individual stations.
- M**oment Tensor –** The Moment Tensor Domain contains classes for applying long period waveform modeling to determine a moment tensor representation of the source of seismic events. The moment tensor quantifies both the event size, and the event type (earthquake vs. explosion).
- **Event Source Determination –** The Event Source Determination Domain contains classes defining tools used by the Analyst to determine the source type that an event (e.g., natural or man-made phenomena).
- **Event –** The Event Domain contains classes for tracking general event information not addressed by other domains. This includes classes for tracking event version history information, finding and tracking events of interest, performing event searches, and computing event quality.
- **Performance Monitor –** The Performance Monitor Domain contains classes for tracking both system performance and monitoring mission performance. The system performance is characterized by disk usage, CPU load, network traffic, etc. The mission performance is characterized by waveform data availability, station signal detection rates, network event detection rates, etc.
- **Signal Feature Prediction** (includes physics models and historical models) **–** The Signal Feature Prediction Domain contains classes for calculating predicted values and uncertainties for observables associate with particular source to receiver paths (e.g., seismic travel time, azimuth, and slowness).
- **Geospatial Processing –** The Geospatial Processing Domain contains classes that access information that identifies the geographic location and characteristics of natural or constructed features and boundaries on the Earth, typically represented by points, lines, polygons and/or complex geographic features, and may contain information attached to a location. Geospatial data is often accessed, manipulated or analyzed through Geographic Information Systems (GIS).
- **Station –** The Station Domain contains classes that define the installation where monitoring sensors are installed. Multiple sensors can be installed at the same station.
- **System Configuration –** The System Configuration Domain contains classes that describe the complete set of system parameters that define the operation of the system software. Examples include sensor thresholds, filters (see filter, waveform), the particular version of an earth model in use and processing sequences Each instance of a system configuration is saved so the state of all parameters at any time can be recalled.
- **Process Control –** The Process Control Domain contains classes that define the configuration and sequencing of processing components in the system.

- **Data Distribution –** The Data Distribution Domain contains classes that provide access and distribution of processing results to external customers of the system.
- **Testing –** The Testing Domain contains classes that create, store, and run system tests, compare system test results to expected results, and test the system via replay.

4.2.2 Analysis Classes

Analysis Classes are the fundamental building blocks of use case realizations. The Use Case Realizations map the system functionality described in Use Cases onto the Analysis Classes. Through this process each Analysis Class contains a high-level description of the functionality the class is responsible to implement. The Analysis Classes in turn become the foundation for organizing the software implementation.

4.2.3 Types of Analysis Classes

The class stereotypes below are used to designate the responsibilities of the analysis classes used in modeling Use Case Realizations. The stereotypes used are **Control**, **Plugin**, **Utility**, **Interface**, **Plugin Interface**, **Boundary**, **Entity**, **Display, Configuration** and **Mechanism**. Numerous other stereotypes are possible at the implementation or source code level (e.g., *proxy*, *adaptor*, *singleton*). At the architectural level, however, the concern is only with describing the essential building blocks of the system at a rather broad, high level. Thus, within the architecture, nearly all Analysis Classes fall into one of the categories below. In the architecture, all classes are categorized into one of the following stereotypes:

- **Control** – A class that operates in an event-driven manner and encapsulates some significant piece of logic, typically application-level logic. **Control** classes may be instantiated as a separate process or set of replicated processes (e.g., for performance). **Control** classes are designed to support separate instantiation but may in fact be combined into processes with other classes (e.g., **Display** classes, other **Control** classes) for performance reasons. The specific mapping of **Control** classes to processes is specified in the Process View.
- **Plugin** – A class that encapsulates an isolated portion of the system which can be updated independently. Plugins may identify portions of the system for which multiple different implementations exist, such as key algorithms. A **Plugin Interface** class defines the common interface for all implementations of **Plugin** class behavior. **Plugin** classes are designed to have simple interfaces to facilitate development and integration of new implementations while limiting the impact to the remainder of the system. To support this relative isolation, a **Plugin** class may only depend on **Plugin Interface**

classes and the Object Storage and Distribution Mechanism (OSD). **Plugin** classes are highly scalable and configurable to meet system performance constraints. **Plugin** classes may be deployed in the same process as the **Control** class using the **Plugin** or in a separate process. The specific mapping of **Plugin** classes to processes is specified in the Process View.

- **Utility** – A class that encapsulates program logic, sometimes at the application-level. Unlike **Control** classes, **Utility** classes are not designed to be instantiated as separate processes; instead, they are designed to be collocated in the same process with **Control** or **Display** classes. Their purpose is to assist that **Control** class in carrying out its function. A given **Utility** class may be used by several **Control** classes. Examples are math libraries and System Clock.
- **Interface** – A class that abstractly represents a defined interface to a class. Unlike **Control** and **Utility** classes, **Interface** classes have no intrinsic behavior built into them; they are simply used to describe messages and data communicated between classes that may potentially be instantiated in different processes. **Control** classes send data to other classes that may exist in separate processes via interfaces by *using* and *realizing* interfaces. In particular, a **Control** class that sends messages according to an interface is said to *use* that **Interface**, while a class that receives messages according to an interface is said to *realize* that **Interface**.
- **Plugin Interface** – A class that abstractly represents the defined interface to a **Plugin** class. A **Plugin Interface** class is similar to an **Interface** class but the **Plugin Interface** class is only realized by a **Plugin** class. Classes invoking a plugin always communicate with the **Plugin** class through a **Plugin Interface** class rather than directly calling operations on the **Plugin** class.
- **Boundary** – A class that abstractly represents an external system or actor (i.e., user). Much like **Interface** classes, **Boundary** classes are abstract and therefore possess no intrinsic behavior. **Boundary** classes can thus be viewed as a special kind of **Interface** class; one in which the sender or receiver is always external to the System. **Boundary** classes may represent any external actor including users, device interfaces, or machine-to-machine interfaces (Transmission Control Protocol/Internet Protocol [TCP/IP], File Transfer Protocol [FTP], etc.).
- **Entity** – A class that encapsulates data rather than logic. **Entity** classes are typically simple classes for holding data. Because their internal state is fully self-contained, they may be persisted in a database or passed between processes as arguments of inter-process function calls.
- **Display** – A class that represents a user interface display. **Display** classes are similar to **Control** classes in that they run in an event-driven manner, and may be instantiated within processes in various combinations (as

specified in the Process View). However, unlike **Control** classes, **Display** classes are dynamically created as needed according to events (e.g., a button press), a distinction important enough to warrant the separate category. **Display** classes also can be started and stopped by an external control process.

- **Configuration** - A class representing a set of related configuration settings defining the default algorithm parameters used when **Control** and **Plugin** classes are invoked. **Configuration** classes support the ability to define parameters based on geographic region, time of year, time of day, network, station, channel, phase, observable type and processing stage. **Configuration** classes contain version information such as installation time, the system release that included the configuration change, etc. **Configuration** classes may be grouped into logical collections (e.g., processing sequence configuration, station processing configuration, location configuration) to organize the settings into general categories. This makes it easier for System Users to navigate the System configuration to find particular configurations. The System Maintainer sets configuration settings offline and installs them on the system.
- **Mechanism –** A **Mechanism** class representing a basic service or framework required by many subsystems across the system. Examples include Inter-Process Communications, Processing Sequence Control, or Object Storage and Distribution—fundamental components which make up the framework upon which the application is constructed.

4.2.4 Control-Based Architecture

A concern in this effort is the requirement to decouple applications so they can be developed and replaced without affecting other parts of the system. Processes within the System may be initiated in two ways: automatically, in response to new or changed data, or interactively, in response to user commands. The system initiates automatic processing to analyze station data, detect signals and group signal detections into events. System users invoke interactive processing to also detect signals and group signals into events. Further analysis is performed to estimate the event location, magnitude, and event source. After interactive processing is completed, the system may initiate further automatic processing to compute other supporting information for the events formed by the operators. Similar analysis is performed in automatic and interactive processing so the system is designed to invoke the same processing components during automatic and interactive processing.

Another design concern is that the system should facilitate the modification or replacement of processing components and permit the processing components to be reordered or linked in alternate sequences. These capabilities are supported by decoupling processing components to the maximum extent and limiting the communication between components. The Processing Sequence

Control Mechanism initiates automatic processing in **Control** classes based on rules defined by privileged users. **Control** classes operate on data stored in the system and store the processing results back in the system, limiting the information passing between **Control** classes.

The approach taken is to employ a concept called Control-Based Architecture (CBA). In essence, CBA dictates that **Control** classes are the logical units of control in the system. In more formal terms, CBA is embodied by the following principles:

- All significant application logic is encapsulated by **Control** classes (not **Utility**, **Entity** or **Display** classes).
- **Control** classes are started and stopped by an external controller program, either the System Control Mechanism or the Processing Sequence Control Mechanism depending on the lifecycle of the **Control** class.
- **Control** classes retrieve inputs from the Object Storage and Distribution Mechanism (OSD), delegate algorithmic computations to **Plugin** classes, and store results in the OSD.
- **Display** classes retrieve data from the OSD.
- **Control**, **Display**, and **Plugin** classes are the only classes that utilize interprocess communication.

Adherence to these principles provides several important benefits, including the following:

- Provides encapsulation of application logic.

  Control classes are designated as the controllers for well-defined portions of application logic (e.g., event location, event magnitude). The Processing Sequence Control Mechanism initiates the **Control** classes and coordinates application logic among the **Control** classes. This approach limits the dependencies between **Control** classes. **Control** classes retrieve data from the OSD, implement processing logic either directly or via delegation to **Plugin** or **Utility** classes, and then return results to the OSD to signal the Processing Sequence Control Mechanism that the processing unit is complete. Encapsulating all of the sequencing responsibility in the Processing Sequence Control Mechanism rather than spreading that responsibility across all the **Control** classes supports the ability to fully configure processing sequences.

- Application logic can be relocated easily.

In essence, each **Control** class represents a relocatable unit of application logic which can be located in any process, as needed. A given process might contain multiple **Control** classes if each has little processing to perform. Alternatively, a process might be dedicated to a single **Control** class to ensure maximum single-processor performance. The ability to relocate application logic as needed is a key capability for providing the flexibility to meet unknown future processing requirements with minimal source code changes.

- Class dependencies are greatly simplified.

  Control, **Display,** and **Plugin** classes perform communication, if required, though Interface classes. Therefore, **Control**, **Display,** and **Plugin** classes never depend directly on one another. Besides facilitating relocatability (as described in the previous bullet), this results in a reduction in dependencies between classes. This reduction in dependencies, in turn, facilitates the ability to reorder or replace processing units.

4.2.5 Use Case Realizations

The process of analyzing use cases (which have an outward focus), and elaborating how the system will accomplish them internally via cooperating Analysis Classes, is a process referred to as *use case realization*, and is the primary activity performed by the Architecture team. The result of this activity is a complete and detailed set of use case realizations covering the entire set of use cases defined for the system.

Use Case Realizations are particularly important in the development of architecture. A Use Case Realization describes the collaboration of analysis classes to implement the functions defined in the Use Cases. The collection of Use Case Realizations in turn define the structure of the architecture needed to implement the significant features of the system. Use Case Realizations also help identify common patterns of interaction which form the basis for defining architectural patterns and mechanisms. Once the Use Case Realizations are developed they become the basis for detailed software design.

4.2.6 Mechanisms

4.2.6.1 System Control Mechanism

In a distributed system it is important to ensure all processes are fully initialized and ready to function before allowing data to flow into the system. Processes may need to interact with other processes to complete initialization. Similarly, when powering down it is important that processes are staged-down in a controlled manner to ensure the system is not left in an inconsistent state.

Because of these issues, a mechanism for coordinating the startup and shutdown of processes across the system is required. This mechanism is known as System Control. This mechanism also monitors processes to ensure they are executing properly.

4.2.6.2 Processing Sequence Control Mechanism

The System provides the capability to execute pre-defined processing sequences for automatic data processing. This capability is supported by the Processing Sequence Control mechanism, which is responsible for managing the execution of processing sequences based on definitions installed in the system. The Processing Sequence Control mechanism and select Processing Sequences are described further in the Use Case Realization reports.

The Processing Sequence Control mechanism executes Processing Sequences based on triggering events in the system. Example triggers include the following:

- **Timer events** – Processing Sequences may be executed at pre-configured times or intervals (e.g., periodically checking for new waveform data to process).

- **Service Invocation** – Processing Sequences may be executed based on invocation of the Processing Sequence Control mechanism's service interfaces (via API or message-based service call). This type of trigger supports execution based on operator commands and other events in the system; for example, for post-processing of created/modified data entities (signal detections, event hypotheses, events, etc.), processing stages, etc.

- **Data Subscription Callbacks** – The Processing Sequence Control mechanism maintains subscriptions for select data updates in the system that require a processing response (e.g., the creation of a new event). These subscriptions and the corresponding Processing Sequence(s) are installed as configuration items in the system. When the Processing Sequence Control mechanism receives callbacks for configured data subscriptions, it invokes the associated Processing Sequence(s).

The Processing Sequence Control mechanism supports a scalable, distributed processing model for execution of processing sequences. Tasks executed by the Processing Sequence Control mechanism may be implemented as service invocations routed to **Control** classes running in separate processes, potentially on separate hosts within the system. This approach allows for parallel execution of Activities within a Processing Sequence across multiple processes and nodes.

4.2.6.3 Object Storage and Distribution Mechanism

A central component of the System is the persistent storage of an extensive history of sensor and station data, station configuration data, significant signal detections and events. In previous versions of the system this information is stored in a Relational Database Management System (RDBMS) and access to the data was achieved by directly interacting with the RDBMS query language. This direct form of access resulted in strong dependencies between the application software and tools and the database structure, resulting in significant impact when modifying or expanding the underlying database schema. A design goal for the modernized system is to isolate the application software from the underlying database schema and data storage technology to limit the impact of changing the database structure and implementation..

This isolation will be achieved through a mechanism called the Object Storage and Distribution Mechanism or OSD. The OSD will be responsible for persisting and retrieving data in the System. Structured Query Language (SQL) calls from software applications will be replaced by calls to the OSD to retrieve the data. The OSD will be responsible for translating the data calls into queries on the underlying data storage technologies, integrating results if the query required data from multiple data stores, and returning results. The data storage technologies could include RDBMS, a waveform store (see section 4.3.4.2.3 for possible technology selections), a provenance store (see section 3.4.2 and section 4.3.4.6), etc. This pattern of abstracting the database interface is known as Data Access Object design pattern. The definition of the query interface to the persisted data and associated data class definitions is described in the Common Object Interface (COI). The OSD provides interfaces using ubiquitous technologies (e.g., HTTP, JSON) to support access from multiple languages and may also provide standard programming language interfaces for a limited language set. Section 4.2.7.1 describes data access implementation patterns. Section 4.3.4.1 describes related executable architecture prototyping work. Information available in the OSD is synchronized between the primary and backup. The synchronization implementation will use a combination of data replication strategies provided by the selected database technologies and custom software.

The OSD will also provide a subscription service to notify any software application that has registered interest in a data object when the data object is modified. Data often needs to be pushed or *distributed* to interested clients (*subscribers*) at the time that it is stored in the database (i.e., *persisted*), a pattern commonly referred to as *publish/subscribe*. The System will distribute either whole **Entity** classes to clients or, if needed to optimize performance, fine grained data distribution at the level of changes to **Entity** classes may be distributed to clients. The OSD may cache data in memory in addition to database storage to decrease access time to retrieve the data. The OSD will

support the typical database functions Search, Create, Retrieve, Update, and Delete (or SCRUD). The Delete operation will be limited to only system administrators. Instead of delete, other operators will be able to mark an object as removed. Removing an object will mark the object as invalid but keep a copy in storage for review or further analysis.

The following figure illustrates the components of the OSD Mechanism developed during prototyping in Iteration E2. The mechanism provides two basic functions: Stored Data Access and Data Distribution. The figure shows the interaction of Data Access Objects with **Entity** Classes and the ORM to provide data storage and access. The figure also shows access to the stored data via scripting languages. In addition, the figure shows Data Distribution implemented via Publish/Subscribe Notifications or via Caching. This model of the OSD Mechanism is the basis for the executable architecture and will evolve from further prototyping.

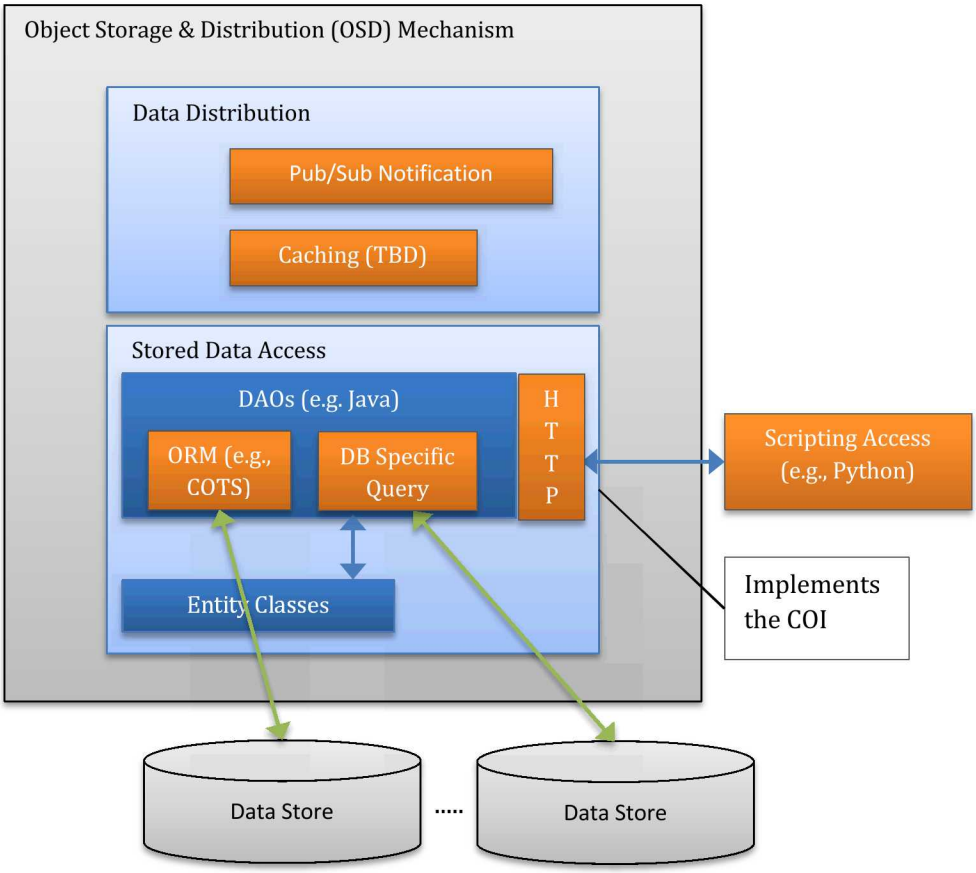

Figure 4-1. OSD Components

4.2.6.4 Inter-Process Communication

As described previously, the OSD Mechanism is the primary means for exchanging data in the System. When communication is required between **Control**, **Display,** and **Plugin** classes, the communication will be implemented via inter-process communication or IPC. Because of the highly distributed nature of the System, limiting the use of IPC will support modifying the processing sequence and replacing **Control** or **Plugin** classes. When used, the IPC mechanism supports patterns of communication, described below:

- **Request/Reply** – A type of call where the caller is not blocked, but can receive return data from the recipient at a later time via a callback. This pattern of communication is used frequently. In particular, **Request/Reply** communication is used heavily for communication between **Display** classes and **Control** classes. In this situation the **Display** class typically makes a request and then displays the result when the reply is subsequently received. This pattern is ideal for the user interface, which cannot afford to be blocked waiting for the reply but also needs to provide confirmation or feedback to the user regarding the request.
- **Publish/Subscribe –** In a publish/subscribe application, senders publish messages to a named topic that serves as a routing key for messages. Consumers may subscribe to one or more publish/subscribe topics in order to receive messages published to those topics. All subscribers to a given topic will receive copies of every message sent to that topic. Publish/subscribe is typically implemented as asynchronous communication where publishing and receiving messages are decoupled.

4.2.7 Patterns

Patterns describe common architectural approaches to addressing various tasks in the system. Patterns describe common interactions between the application software and the basic mechanisms. These patterns emerge from development of the Use Case Realizations and provide some of the fundamental building blocks for the system architecture.

4.2.7.1 Data Access

Control classes interact with the Object Storage and Distribution Mechanism to access persistent data in the system. The OSD Mechanism implements *get* and *store* operations that provide access to data. The OSD Mechanism abstracts the underlying data persistence technology and query language, presenting clients an implementation agnostic programming interface. The **Control** class is dependent on the OSD but not dependent on the underlying database. The **Control** class passes the attributes identifying the object to retrieve to the OSD

Mechanism and the requested object is returned to the **Control** class. The **Control** class modifies the requested object and then calls the OSD Mechanism to store the object. The OSD Mechanism also provides a local store operation that updates the object but the update is not visible globally. The selection for storing globally or locally is defined in a Processing Context object that is passed to the OSD Mechanism as part of the store operation. The OSD Mechanism can also notify **Control** or **Display** classes when an individual object or collection of objects is modified. **Control** or **Display** classes register a callback for the object or collection of interest with the OSD Mechanism and the OSD calls the callback when the object or collection is modified. A common example occurs when a **Display** class provides a list of events in the system. The **Display** class registers a callback with the OSD Mechanism for all events. The OSD Mechanism calls the **Display** when new events are created or modified and the Display class updates the user interface. **Control** or **Display** classes unregister when the notification is no longer required.

For subscription-based OSD data callbacks that occur over a network (e.g., between processes and/or nodes), the executable architecture prototype will use a pattern similar to the *Claimcheck* Enterprise Integration Pattern (EIP).[1] Rather than serializing and transmitting the data **Entity** to subscribing processes directly, under this pattern, the OSD will store the **Entity**, making it globally accessible, and will transmit a "claimcheck" message to subscribers. The claimcheck message will include reference information sufficient for the subscriber to retrieve the **Entity** from the OSD upon receipt of the message. This pattern provides an efficient means of communicating data between processes, even for large data entities.

4.2.7.2 Processing Context

The Processing Context object describes to the OSD and PSC mechanisms why data is being stored and processed. A Processing Context includes information about how data was created and also includes a storage context specifying whether the data is stored with a private visibility or with a global visibility. The OSD manages data access based on its storage context. Data stored with private visibility is only accessible within the Processing Stage that stored the data. This Processing Stage can be one of the System's automatic Processing Stages or a Processing Stage corresponding to an Analyst's interactive analysis session. When a Processing Stage stores data to the OSD with a private context it allows the OSD to distribute that data to Display classes for presentation and to the PSC mechanism or Control classes for processing while keeping the data's visibility local to that Processing Stage. When a Processing Stage stores data with a global context the OSD makes that data globally accessible and available for display and processing by other Processing Stages. Since data stored with a private context is only accessible within the Processing Stage storing the data, this type of

[1] G. Hohpe and B. Wolfe, Enterprise Integration Patterns, Boston: Addison-Wesley, 2003.

storage is temporary. The OSD Mechanism may remove the data at the end of the Processing Stage. Data stored in a global context is persisted indefinitely in the OSD Mechanism for future access. Processing Context also contains creation metadata describing how the data being stored was created. This metadata includes the automatic or interactive Processing Stage creating the data. When applicable, the metadata also references the Processing Step and Processing Sequence being executed when the data was created.

4.2.7.3 Analysis Modeling for User Interface

Display classes are the primary point of interaction between the user and the System. They model how the System presents information to the user and how the user provides input to and requests actions on the System. **Display** classes are typically opened when requested by the user and closed when the user has completed the interaction. **Display** classes request and receive data from the OSD either by directly requesting information or subscribing for updates. **Display** classes synchronize their views with changes made to the underlying data model. **Display** classes also pass user requests to **Control** classes through IPC.

The design of the Analyst workspace is independent but related to Use Case Realization **Display** classes. The **Display** classes for Analyzes Event UCR, Refines Event URC, Scans Waveforms and Unassociated Detections UCR, and several other children UCRs define subsets of information and interactions that are part of the workspace. For example, the Analyzes Event Display is responsible for displaying the list of events, the Selects Data for Analysis Display displays the operator workflow, the Refines Event Display shows the detailed information about an individual event, the Enhances Signals Display shows the waveforms during the time interval of an event, and the Detects Signals Display shows the signal detections associated with an event.

In Use Case Realizations, **Display** classes are primarily modeled one per use case to capture a summary of the information transferred via the user interface for that use case. **Display** classes are also modeled to reflect common user interface functionality shared between Use Case Realizations. **Display** classes do not represent the actual layout of screens or windows in the user interface, and are presented as a means of communicating the interactions between users, **Display** classes, **Control** classes, and **Mechanisms** (like the OSD). The design and layout of **Display** classes along with input action behavior are addressed in User Interface Storyboards. Storyboard mockups aim to provide a visual representation of the flow of action between a user and the System for a given Use Case.

4.2.7.4 Algorithms for Automatic and Interactive Processing

The System has the requirement to allow the users to access the same algorithms used during automatic processing. This is achieved by the Processing Sequence Control Mechanism invoking the **Control** class for automatic processing and a **Display** class invoking the same **Control** class for interactive processing. The System Maintainer sets the default **Configuration** used during automatic processing while Analysts can select to override the settings used during interactive processing. When first started by System Control both **Control** and **Plugin** classes use **Configuration** classes to determine their default settings. The **Control** and **Plugin** classes will use these defaults for any settings that are not explicitly overridden in a particular call to the **Control** or **Plugin**. The system associates processing results (e.g., events, signal detections) with the settings used to create those results. Where applicable, the system uses these settings rather than the system default **Configuration** when invoking additional automatic processing to further refine the results. This ensures Analyst settings supersede default settings.

A further requirement is for the system to facilitate the update and replacement of algorithms. The architecture approach for this capability is to identify likely candidates for update and replacement and to model these candidates using **Plugin** classes. The **Plugin** class is invoked by a **Control** class through a **Plugin Interface** class. This isolates visibility of the **Plugin** class from the rest of the system. Figure 4-2 shows the relationship between **Control**, **Plugin Interface**, and **Plugin** classes. The **Plugin** class does not interact directly with other **Control** or **Display** classes in the system but operates on data provided via the **Plugin Interface** class. This limits requirements on the design of the **Plugin** class from implementing the control logic and data access logic developed for the entire system. **Control** classes can select among multiple **Plugin** classes that implement the same function by designating a configuration parameter to identify the desired **Plugin**.

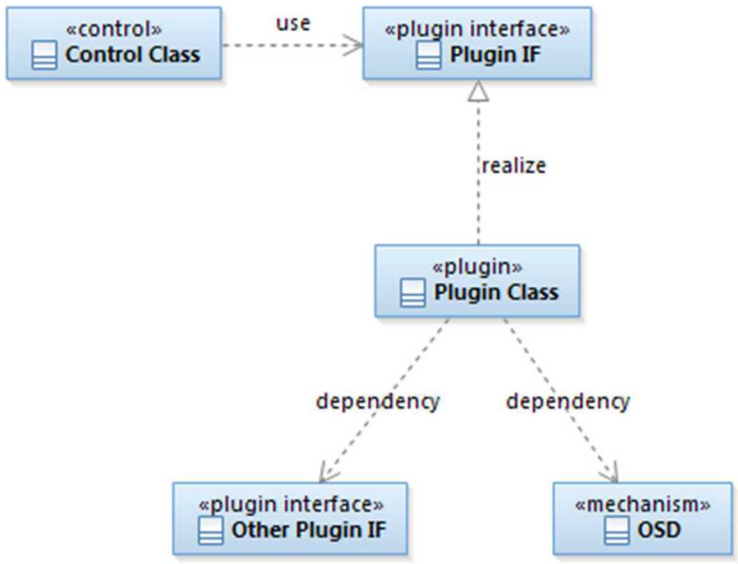

Figure 4-2. Relationship between Control Class and Plugin Class

The following sequence diagrams describe the pattern of interactions between **Control** classes and **Plugin** classes. This pattern is repeated in Use Case Realizations where Plugins are used. All the details of the interactions may not be repeated in the individual UCRs when they do not differ from the pattern described here.

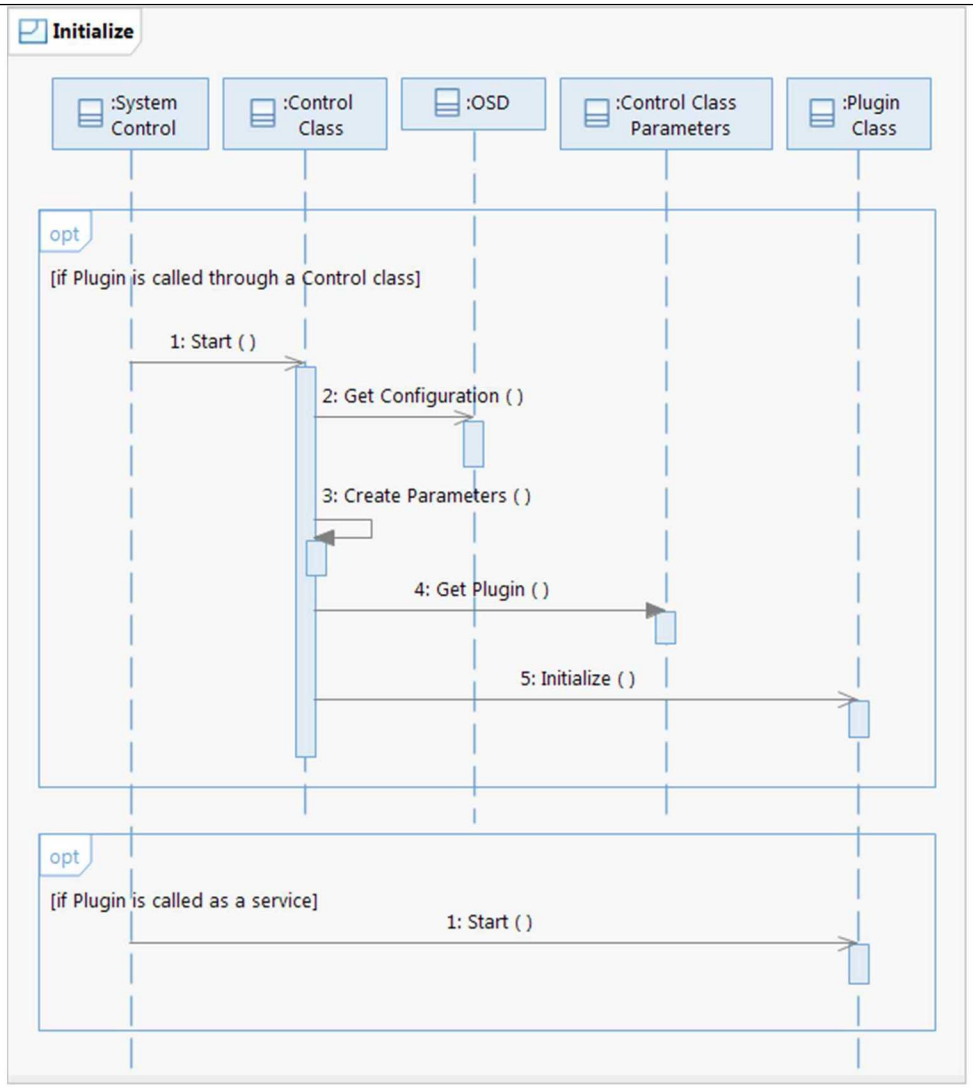

Figure 4-3. Plugin Initialization

This sequence diagram shows how **Plugin** classes are initialized. If the **Plugin** class is called by a **Control** class, the **Control** classes initializes the **Plugin** classes when the System starts. The System Control mechanism calls each **Control** class to initialize. The **Control** class retrieves the default configuration from the Object Storage and Distribution Mechanism and creates the parameters that will be used to call the **Plugin** classes. The **Control** class also identifies the set of **Plugin** classes that implement the **Control** class functionality from the parameters and calls each **Plugin** class to initialize. If the **Plugin** is called as a service, the **Plugin** class is started by the System Control Mechanism.

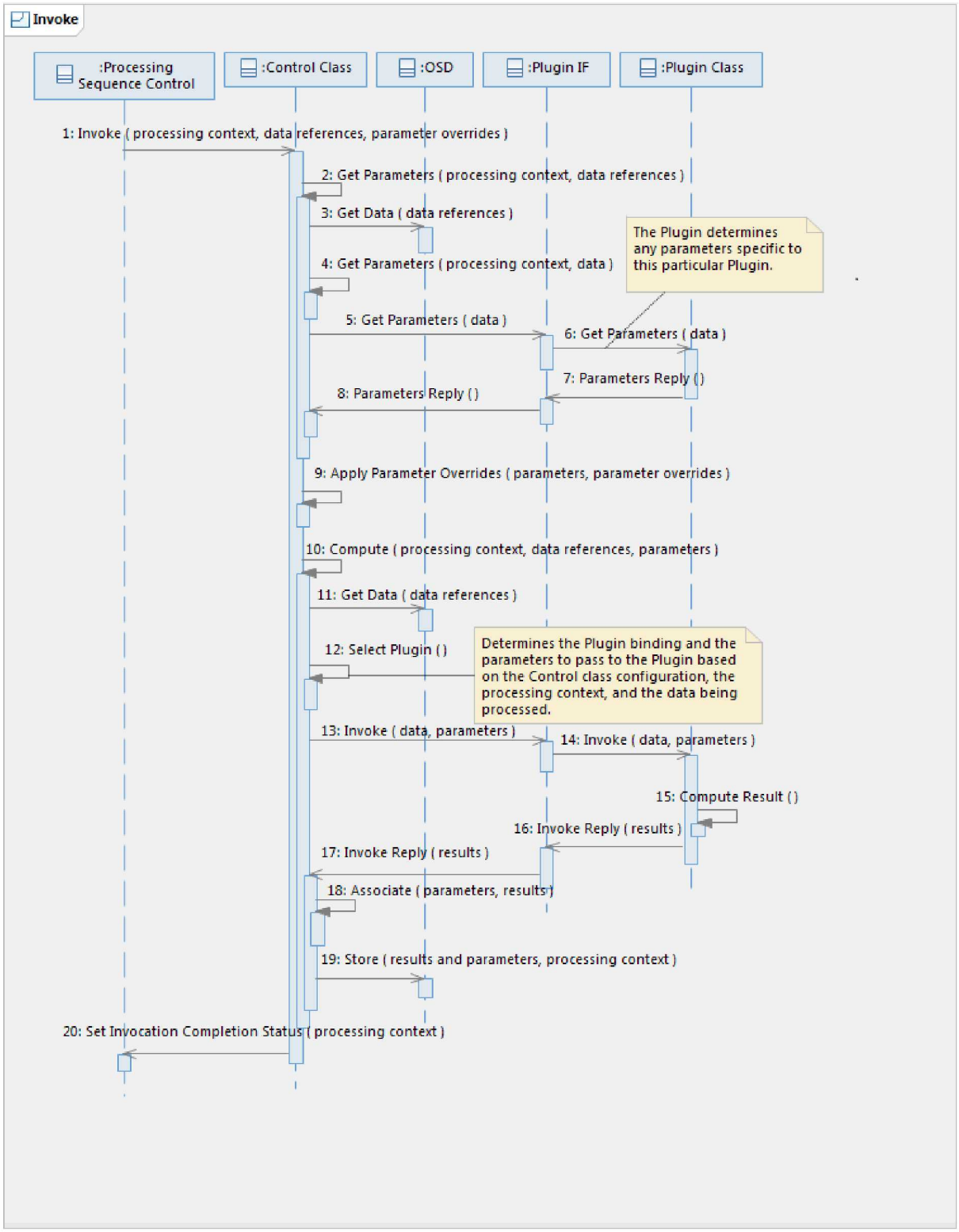

Figure 4-4. Plugin Invocation in Automatic Processing

This flow describes how **Control** classes invoke Plugin classes to execute algorithms or implement models during automatic processing. The Processing Sequence Control Mechanism calls the **Control** class when triggering criterion are satisfied with data references for the information to be processed and the processing context identifying the processing stage. The **Control** class uses the data references to retrieve the data to process from the OSD. The **Control** class determines the **Plugin** class to call and the parameters from the processing context and the data. If the **Plugin** class requires any plugin-unique parameters, the **Plugin** class determines those parameters from the data. The **Control** class sets creation information (e.g., the Processing Stage and system process invoking the **Control** class, invocation time) in the parameters class. The **Control** class calls invoke passing the data and the parameters through the **Plugin Interface** class to the **Plugin** class. By exception to this pattern, if the **Plugin** class requires large data sets that would be inefficient to pass to the **Plugin** class each time it is invoked the **Plugin** class may access either the OSD or a plugin specific data store to retrieve data. The **Plugin** class then computes the result and passes the result back to the **Control** class. The **Control** class then associates the processing parameters with the results, stores the parameters and the result based on the processing context, and notifies the Processing Sequence Control Mechanism of the completion status.

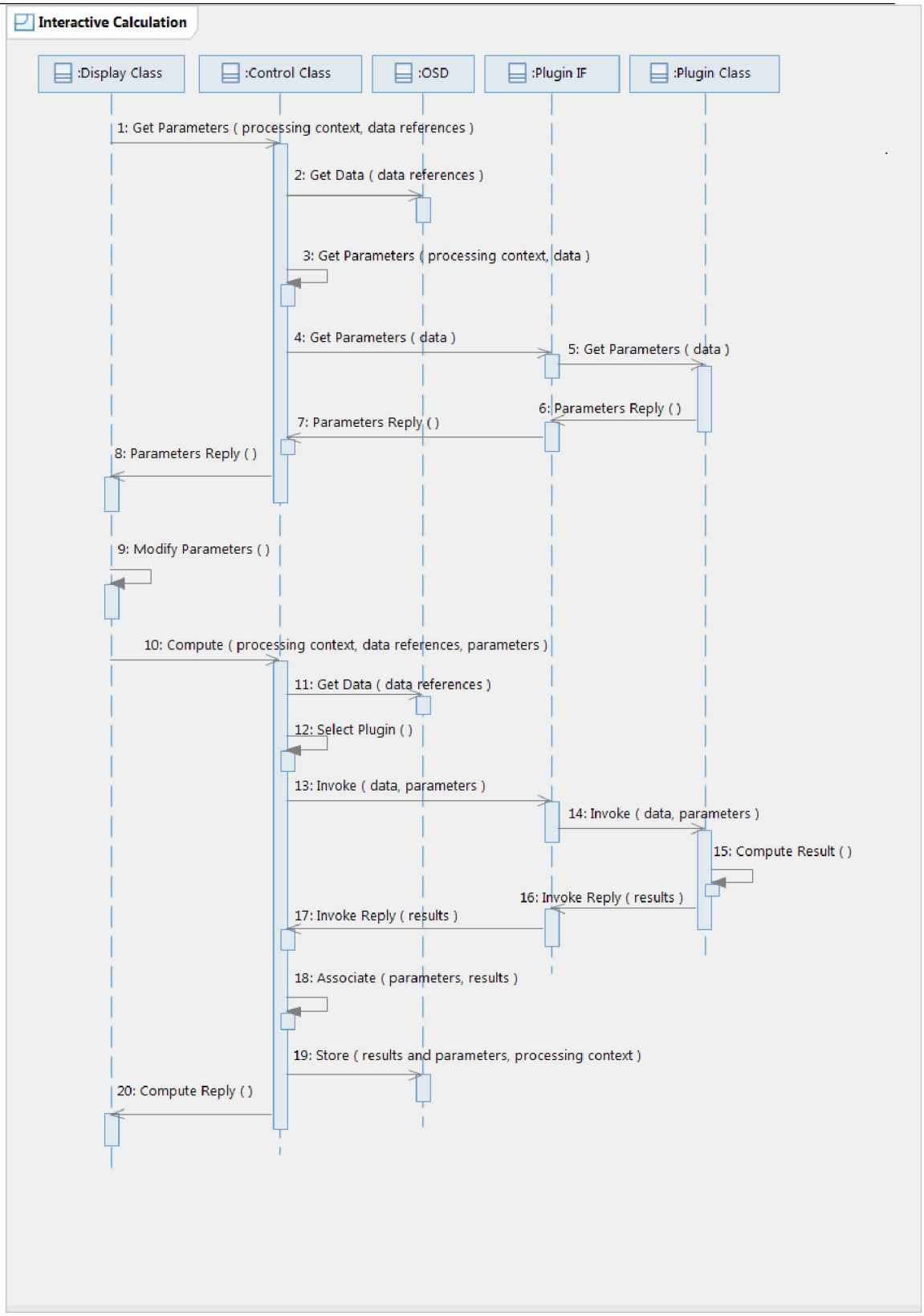

Figure 4-5. Plugin Invocation in Interactive Processing

This sequence diagram shows how **Control** classes and their associated **Plugin** classes are invoked interactively. The **Display** class retrieves the default or last used parameters from the **Control** class based on the processing context and the data. The **Control** class groups the parameters for the **Control** class with the Plugin-unique parameters from the appropriate **Plugin** class and returns the parameter set to the **Display** class. The **Display** class presents the parameter values to the user and the user modifies selected values and requests the System to compute based on the modified set of parameters. The **Display** class calls the **Control** class with the parameters and the **Control** class invokes the appropriate **Plugin** class passing the parameters similar to automatic processing. The **Control** class sets creation information (e.g., the Processing Stage and identifier of the user invoking the **Control** class, invocation time) in the parameters class, associates the parameters with the result, stores both in the OSD, and passes the result back to the **Display** class.

Plugin classes may create results containing information relevant to provenance, performance monitoring, tuning, etc. specific to the plugin implementation that go beyond what the **Plugin** is required to return by the **Plugin Interface**. Since the implementation specific results cannot be known a priori, each **Plugin** is responsible for defining **Entity** classes representing the plugin specific results and associating them with the results returned via the **Plugin Interface**.

Example **Plugin Interface** classes are shown in several UCRs (e.g., System Refines Event Location, System Detects Signals, and System Builds Events using Signal Detections). Most of the **Plugins** perform calculations but some of the **Plugin** classes implement a **Plugin Interface** designed to provide data used in other calculations (e.g., the Earth Model Plugin provides values to Signal Feature Predictor Plugins). The table below lists the **Plugin Interface** classes that are currently in the Analysis Model. In some cases, the Analysis Model also includes particular specializations of a **Plugin Interface** that may exist in the System to satisfy system requirements while in other cases the Analysis Model only includes the basic **Plugin Interface**.

Table 4-1. Modeled Analysis Plugins

Plugin Interface	Plugin Specializations	Defined in UCR	Used in UCR
Signal Detector		System Detects Signals	System Detects Signals, Detects Signals
Signal Onset Time Refiner	AIC Signal Onset Time Refiner, Waveform Cross-Correlation Signal Onset Time Refiner	System Detects Signals	System Detects Signals, Detects Signals
Signal Detection Associator	Match Signal Detection Template	System Builds Events using Signal Detections	System Builds Events using Signal Detections, Builds Event (UCR not modeled)
Waveform Correlation Event Detector		System Detects Events using Waveform Correlation	System Detects Events using Waveform Correlation, Builds Event (UCR not modeled)
Event Locator	Master Event Locator	System Refines Event Location	System Builds Events using Signal Detections, System Refines Event Location, Refines Event Location
Signal Feature Predictor		System Predicts Signal Features	System Predicts Signal Features, System Builds Events using Signal Detections, System Refines Event Location, Detects Signals (not modeled in UCR), System Refines Event Magnitude (UCR not modeled), Refines Event Magnitude(UCR not modeled), Monitors Mission Processing (UCR not modeled)
Earth Model		System Predicts Signal Features	System Predicts Signal Features, System Builds Events using Signal Detections, System Refines Event Location, Detects Signals (not modeled in UCR), System Refines

			Event Magnitude (UCR not modeled), Refines Event Magnitude(UCR not modeled), Monitors Mission Processing (UCR not modeled)
Meteorological Data Update		System Acquires Meteorological Data	System Acquires Meteorological Data
Waveform Data Quality		System Determines Waveform Data Quality	System Determines Waveform Data Quality
—END OF TABLE—			

Additional **Plugin Interface** classes will be created as UCR modeling progresses. The following table lists plugins that may be modeled in the future. This table does not show all plugins that are unique to a particular organization.

Table 4-2. Potential Analysis Model Plugins

Plugin Interface	Plugin Specializations	Defined in UCR	Used in UCR
Multiple Event Locator		System Refines Event Location	System Refines Event Location, Performs Multiple Event Location
Signal Feature Measurer		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
FK Feature Measurer		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
Polarization Feature Measurer		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
Waveform Filter		System Enhances Signals	System Enhances Signals, Enhances Signals
Waveform Rotator		System Enhances Signals	System Enhances Signals, Enhances Signals
Waveform Beamer		System Enhances Signals	System Enhances Signals, Enhances Signals

Moment Tensor Evaluator		System Evaluates Moment Tensor	System Evaluates Moment Tensor, Evaluates Moment Tensor
Magnitude Estimator	Network Magnitude Estimator, Station Magnitude Estimator, Relative Magnitude Estimator	System Refines Event Magnitude, Refines Event Magnitude	System Refines Event Magnitude, Refines Event Magnitude
Phase Labeler		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
Event Conflict Resolver		System Resolves Event Conflicts	System Resolves Event Conflicts
Similar Event Finder		System Finds Similar Events	System Finds Similar Events, Compares Events
Analyst Performance Metric Calculator		Views Analyst Performance Metrics	Views Analyst Performance Metrics
Event Quality Calculator		System Builds Events using Signal Detections	System Builds Events using Signal Detections
Station Performance Calculator		System Builds Events using Signal Detections	System Builds Events using Signal Detections, System Refines Event Location
Network Performance Calculator		System Builds Events using Signal Detections	System Builds Events using Signal Detections
Event Comparer		Analyzes Mission Performance	Analyzes Mission Performance, Compares Events
Bulletin Comparer		Analyzes Mission Performance	Analyzes Mission Performance
Data Provenance Analyzer		Views Event History	Views Event History, Analyzes Research Events
Waveform Correlator		System Detects Events using Waveform Correlation	System Detects Signals, System Detects Events using Waveform Correlation, System Finds Similar Events

—END OF TABLE—

4.2.7.5 Event Analysis Classes

Events are the fundamental output of the System. The following diagram shows the relationships between Events, Signal Detections and Waveforms. An Event is the occurrence of some source of energy within the Earth's body, oceans, or atmosphere that can be detected by seismic, hydroacoustic, and/or infrasonic sensors. The Event class is composed of a set of Event Hypotheses each representing a different analysis of the Event. Each Event Hypothesis contains a summary of the contributing stations, associated signal detections, and a set of location solutions for the Event along with the parameters used to analyze the Event. One of the Event Hypotheses is the overall preferred version of the Event and represents the best available analysis of the Event. A Waveform is either the raw or derived output of seismic, hydroacoustic, and/or infrasonic sensors. A Signal Detection is a signal of interest. Similar to the relationship between Event and Event Hypothesis, each Signal Detection is composed of a set of Signal Detection Hypotheses representing different ways of analyzing the signal of interest. Each Signal Detection Hypotheses is described by a time interval on a Waveform. There may be signal enhancement techniques applied to the Waveform to help reveal the signal of interest. An Association object represents the relationship between an Event Hypothesis and a Signal Detection Hypothesis. An Event Hypothesis and Signal Detection Hypothesis that are associated to each other will each have a relationship to the Association class. If the Event Hypothesis and Signal Detection Hypothesis are later unassociated then their relationship to the Association class is retained for provenance. Event Hypotheses and Signal Detection Hypotheses record the history of the analysis of Events and Signal Detections and thus retain a significant portion of provenance in the system.

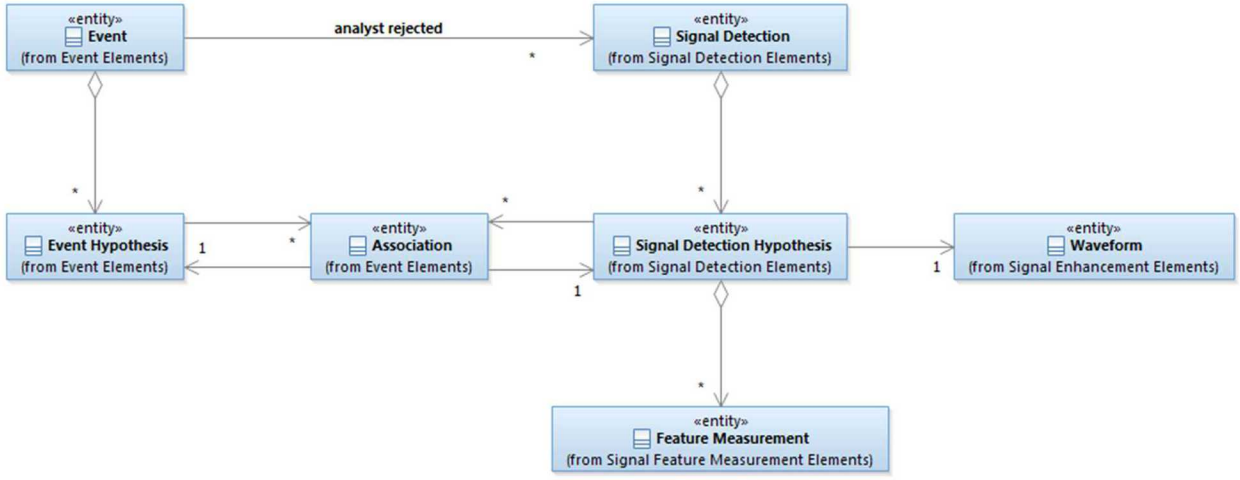

Figure 4-6. Event Analysis Classes

4.2.7.5.1 Event Hierarchy

One of the significant features of the System is retention of intermediate results during event analysis. The previous system contained a limitation that only a single Event Hypothesis could be stored for each stage of automatic or interactive processing. The System will allow Analysts to store multiple Event

Hypotheses during each processing stage as shown in the figure below. Because multiple Event Hypotheses may be saved for each processing stage, the Analyst must designate which Event Hypothesis is the preferred version for that processing stage. Another significant contributor to event analysis is the set of data available to each Analyst at the time they are reviewing the event. Late arriving data can modify the solution so a record of the data available at the time an Analyst is reviewing the Event is an important factor in understanding the event solution result. Providing the complete set of Event Hypotheses that compose an Event allows detailed post analysis of the event formation process.

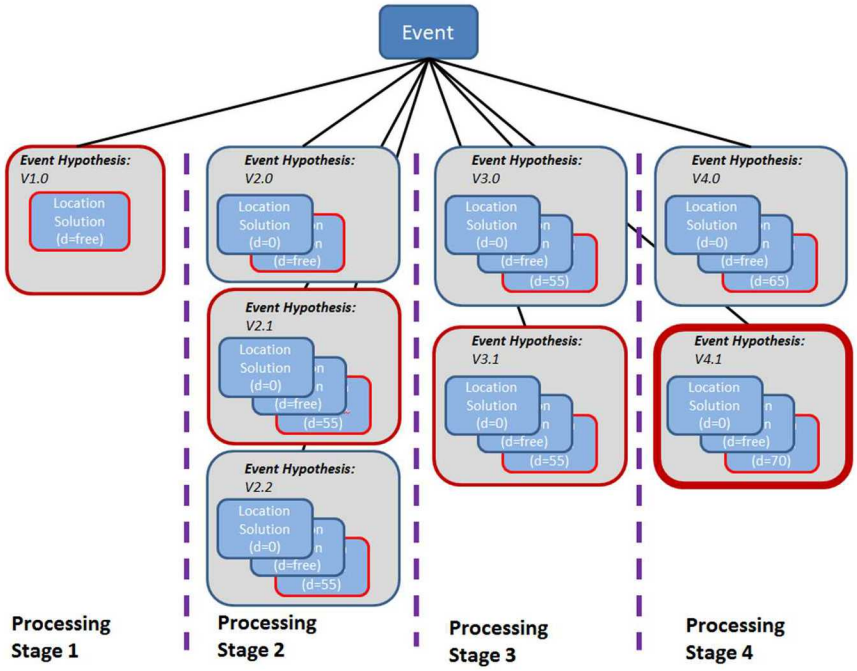

Figure 4-7. Event Hierarchy

4.2.7.5.2 Processing Stages

Events progress through several levels of analysis by both the System and Analysts. The various levels of analysis are identified by the term Processing Stage. The ability to modify and define the set of Processing Stages and Processing Sequences is a requirement for the System. To facilitate this goal, the Processing Stages and Processing Sequences associated with each stage are defined as part of the configuration of the system. The Processing Sequence Control Mechanism will control the overall System processing, calling the Processing Sequences associated with a Processing Stage based on the configured triggers. The Analysts control the progression from one Processing Stage to another.

Typically, as data arrives at the System, the System processes input data and creates a set of initial Events as the first processing stage. The System will continue to process waveform data as it arrives and automatically refine Events

until an Analyst opens a time interval for interactive analysis in a new Processing Stage. When the Analyst completes their analysis the System enters a system Processing Stage where the System updates Events with any late arriving data and updates any Event inconsistencies left from the Analyst Processing Stage. Events continue to progress through the configured series of Processing Stages until reaching a state where the Event information is released to external customers. The System captures information detailing the evolution of each Event as it proceeds through the Processing Stages to provide provenance for each Event.

4.2.7.5.3 Error Handling and System Logs

Common methods of error handling and message logging will be defined for the system and all system components including algorithm implementations will adhere to the common methods. The intent is to utilize standards and off-the-shelf components to implement these functions. The message logging implementation should have the capability to aggregate all system messages in a single log as an aid in debugging.

4.3 Implementation View

The Logical View is concerned with defining the Analysis Classes and the collaboration between them necessary to support the use cases. The Implementation View, on the other hand, is concerned with organizing those Analysis Classes into related groups to facilitate development. In practical terms this entails grouping Analysis Classes together into modules, each of which produces a single library, followed by further groupings of subsystems into layers. Several factors bear consideration when determining these groupings, including the following:

- Classes that work closely together should be grouped together into the same subsystem to hide complexity from outside clients.
- Classes that are logically independent of each other should be kept in separate subsystems so that clients get only what they really need when they use a subsystem.
- Classes that require similar developer domain expertise should be grouped together, to facilitate team development.
- Dependencies between subsystems should be minimized, to the extent possible, to facilitate parallel development.
- Circular dependencies are not permitted.

4.3.1 Layers

The System software is organized into layers, as depicted in Figure 4-8. Implementation View Layers below.

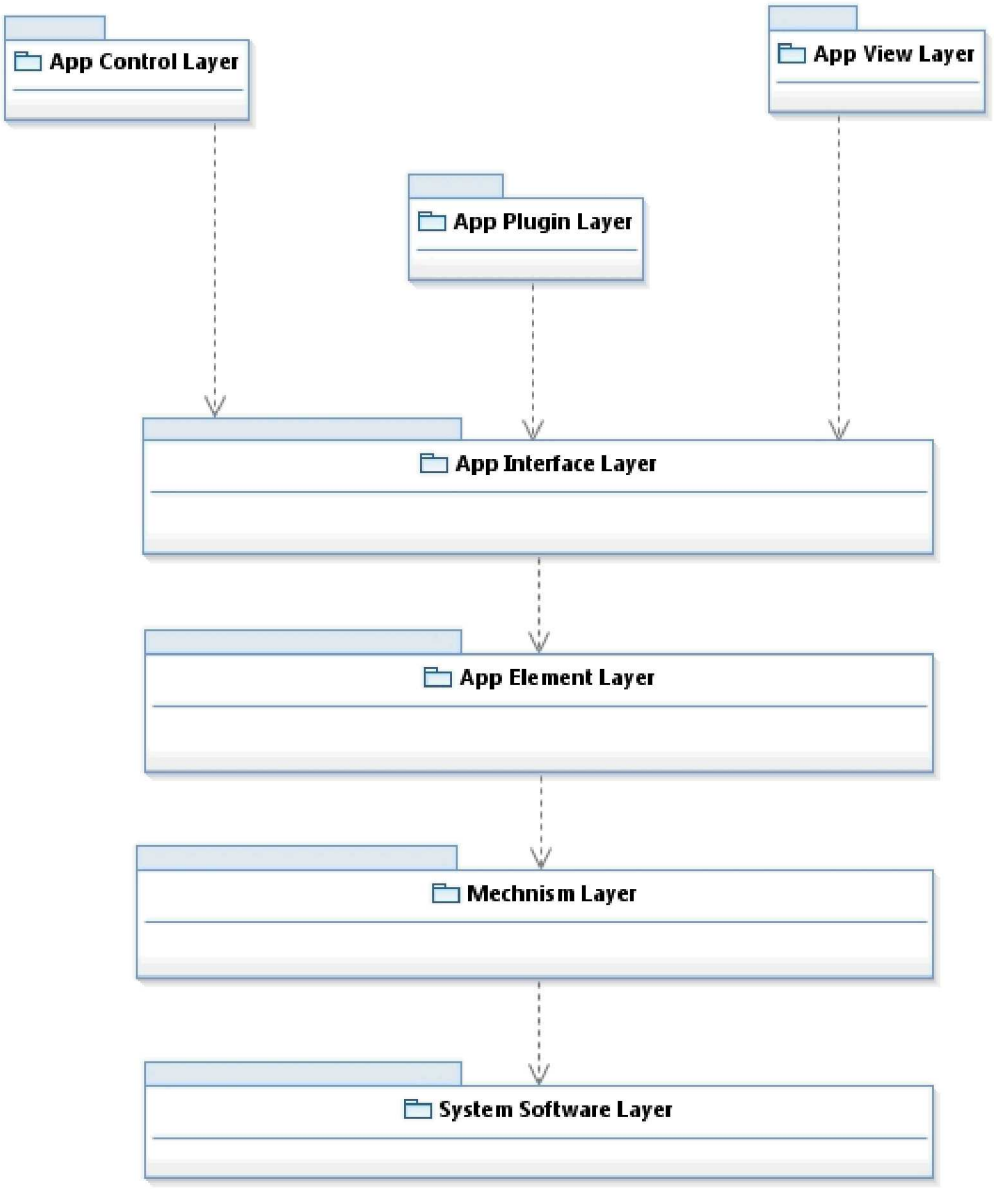

Figure 4-8. Implementation View Layers

Each layer contains numerous subsystems, which in turn contain the actual source code files. Subsystems may depend upon (or *import*) other subsystems according to their needs, but only so long as they adhere to the following rules:

- In general, a subsystem in a given layer may only depend on another subsystem if it is within the same layer or below.
- Import cycles are never allowed (e.g., A imports B, B imports C, and C imports A).

Adherence to these rules has a minimizing effect on dependencies in the system overall, resulting in software that is easier to develop, understand and maintain. Each layer is intended to encapsulate the system at a particular level, as described below:

- **System Software Layer** – Provides direct access to COTS, open- source software and third-party libraries. These include, among others, hardware device drivers and the OS.
- **Mechanism Layer** – Contains subsystems that implement basic services required by many subsystems across the system. Examples include Common Object Interface, Processing Sequence Control, and System Control — services which provide the framework upon which the application is constructed. It also includes primitive classes such as strings, times, timers and mathematical elements. Because this layer is intended to isolate the rest of the application from specific COTS or open-source software and OS implementations (thus improving portability), application code in layers above are expected to use the mechanisms provided in this layer rather than using COTS, open- source software and OS features directly whenever possible.
- **Application Elements Layer** – Contains only **Entity** and **Utility** classes. This layer encapsulates the data elements of the application (e.g., Events, Signal Detections) which are needed by multiple subsystems. Because these classes are reused extensively, it is particularly important to minimize dependencies between subsystems in the **Application Elements Layer**. Subsystems in this layer are organized into a dependency hierarchy, with no cyclic dependencies between subsystems. A goal of this layer is to structure the data to minimize unneeded dependencies in higher layers. For instance, data that is widely used should be as low as possible in the hierarchy.
- **Application Plugin Layer** – Contains **Plugin** classes that encapsulate replaceable portions of the system (portions of the system for which multiple different implementations exist, such as key algorithms). **Plugin** classes are controlled by **Control** classes and are invoked via a **Plugin Interface** class.
- **Application Interfaces Layer** – Contains only **Interface** and **Plugin Interface** classes. This layer encapsulates the inter-process interfaces through which **Control**, **Display** and **Plugin** classes communicate with one another. **Interface** and **Plugin Interface** classes can only depend on **Entity** classes in the layers below (**Application Elements Layer** and **Mechanism Layer**).

- **Application Control Layer** – Contains the **Control** classes that encapsulate application logic. **Control** classes never depend on other **Control** classes. **Control** classes communicate data primarily through the OSD and the Processing Control Mechanism controls the sequence of **Control** class execution. This approach attempts to eliminate coupling between **Control** classes to permit multiple implementations of algorithms and facilitate replacement or addition of new algorithms. If cases arise where direct communication is required between **Control** classes then they would communicate with each other via IPC (**Interface** classes, from the **Application Interfaces Layer**). Thus, subsystems within this layer are all independent from one another (from a static dependency standpoint).
- **Application View Layer** – Contains **Display** classes that encapsulate the user interface. Dependency-wise this layer is logically at the same level as the **Application Management Layer** and thus could have been included as part of that layer. It was separated however because its purpose is different than the **Application Management Layer**; from a client/server perspective the displays in the **Application View Layer** act as the clients whereas the **Control** classes in the **Application Management Layer** act as the servers.

4.3.2 Frameworks

The System architecture is investigating frameworks to leverage open-source or commercial products that could potentially provide key components of the system, especially for user interface generation. Frameworks can provide significant functionality but also impose additional constraints on the implementation. Therefore, prototyping potential framework candidates is essential to determine the usefulness as well as the constraints associated with each candidate.

4.3.3 Executable Architecture Prototyping Goals

The prototyping targets early implementation of key aspects of the architecture. The purpose of prototyping the executable architecture is to demonstrate an implementation of key system features and define fundamental software patterns with the intent of proving the feasibility of the architecture design.

Previously the prototyping team surveyed possible COTS and open-source components and developed exploratory prototypes addressing the following core elements of the system architecture:

- *Inter-Process Communication (IPC)* is a key underlying capability required in distributed computing architectures for the exchange of data, control, and status information between independent processing residing on one or more hosts.

- The *Object Storage and Distribution (OSD)* Mechanism provides application programming interfaces (APIs) for access to data stored in the database as well as distribution of data among processing components.
- *Processing Control* is the software infrastructure that provides for the definition, configuration, execution and control of processing components, supporting both automated and interactive analysis processing. This element supports the System Control and Processing Sequence Control (PSC) Mechanisms.
- The *User Interface Framework (UIF)* provides a flexible platform for the definition of extensible graphical user interface (GUI) components and composition of GUI displays supporting users of the System and research tools.

An important goal for the Executable Architecture is the definition of control flow in the System. Control flow is exercised through the Processing Sequence Control Mechanism (PSC) as described in Section 4.2.6.2, Processing Sequence Control Mechanism. The Executable Architecture demonstrates how the PSC interacts with **Control** classes and their associated **Plugins** to control processing in the automatic pipeline and during interactive processing. The PSC passes control to the configured **Control** class using messages. Data references are passed to the **Control** class using the Claimcheck pattern. The **Control** classes use installed configuration to determine the particular **Plugin** class to invoke and determine the default parameters for the **Plugin** to execute. The pattern for **Control** class / **Plugin** interaction is shown in Section 4.2.7.4 Algorithms for Automatic and Interactive Processing. After execution the **Control** class returns control flow to the PSC Mechanism for the processing sequence to continue.

Another goal of the Executable Architecture is demonstration of how **Control** classes and **Plugins** access data from the OSD. **Control** classes use data references passed by the PSC to call a service interface provided by the OSD to retrieve data assuming a service interface can respond within timeliness requirements. The data is serialized and returned to the **Control** class. Because serialization protocols are supported across multiple languages, the serialization technique provides a method of interfacing software components implemented in multiple languages. Another aspect of the OSD is the capability of **Control** classes to subscribe for updates when data changes. In the current design the OSD is responsible for accepting subscriptions about individual data or collections of data and then using a callback to inform the **Control** class of changes. Prototyping permits testing how this approach would scale to handle the data rates and processing distribution of the System.

The data access prototyping requires a more complete definition of the data objects in the system. Development of the Data Model has begun the definition

of how information is organized in the system. The prototyping effort focuses on how the data is created, transmitted and stored in the System. Prototyping introduces additional constraints on the Data Model which helps mature the Data Model design and inform the development of an example Component Interface Specification.

4.3.4 Executable Architecture Prototype Elements

The Executable Architecture prototyping is currently focused on implementing automated signal detection processing software. As the prototyping progresses, the Executable Architecture will expand to implement interactive signal detection and the supporting user interface component. The prototype will address a number of signal detection-related processing functions, including waveform quality control (QC), signal enhancement (including filtering, rotation and beaming), signal detection and signal feature measurement.

In order to meet the goals described in Section 4.3.3, the executable architecture prototype will include the following key elements:

- Inter-process Communication Implementation
- OSD and Data Model Implementation
- Processing Sequences

The following sections provide a description for these elements, including a discussion of key design decisions and software technologies selected as part of the prototyping work.

4.3.4.1 Inter-process Communication Implementation

The team has investigated two technology pairs that together support the messaging and data encoding required for Inter-Process Communication. Both technology pairs are language-agnostic in order to support applications written in any of the languages selected for the System (i.e., C, C++, Java, and Python).

1. HTTP / JSON – Using this technology pair, clients communicate with external applications by sending HTTP request messages with JSON-encoded bodies containing the request details (e.g., data references, parameters to the external application, etc.). The target application hosts the requested HTTP interface (e.g., via an embedded HTTP server) and responds to requests with an HTTP response message including an HTTP status code and any response data in a JSON-encoded body.

2. AMQP (RabbitMQ)/ JSON– Using this approach clients communicate with external applications by sending request messages via a dedicated AMQP

queue with a JSON-encoded request details (e.g. data references, parameters to the external application, etc.). The target application listens for request messages on the specified queue and responds to requests on a separate response channel (identified in the request) with a response message including processing status and results in JSON.

The primary advantage of HTTP is that it is a ubiquitous solution (used throughout the web) with an enormous development community and user base. As a human-readable format, JSON also facilitates user access to data. The primary limitation of this approach is that HTTP is not well suited to patterns other than request/response (e.g., publish/subscribe). The primary advantage of AMQP is that it supports a number of messaging patterns (direct, fanout, publish/subscribe, request/response, etc.). The primary limitation of this approach is AMQP is less-widely used than HTTP and is more cumbersome for request/response patterns (requiring a request queue and separate response queue identified in the request message).

Based on prototyping efforts for request/reply patterns in the system (e.g. network access to stored data via the OSD, invocation of services) we will use HTTP. For publish/Subscribe (e.g. notification and event driven processing) we will use an AMQP brokered messaging solution. For both of these protocols the system will use JSON to encode the data if the communication meets latency requirements. If necessary, future work will evaluate encoding alternatives as needed to improve messaging latency performance.

4.3.4.2 OSD and Data Model Implementation

As described earlier in the Logical View section, the Object Storage and Distribution (OSD) Mechanism (section 4.2.6.3) provides access to persistent data for client applications implemented in the languages of the System, while insulating these applications from the details of the underlying data storage solution. Key design goals driving the implementation of OSD include the following:

- OSD data access support should be available in multiple development languages, including minimally those selected for the System: C, C++, Java, and Python.

- OSD data access interfaces should be independent of any particular storage paradigm (relational, key-value, column family, etc.), and any particular solution (e.g., Postgres, Oracle). The motivation is to minimize the impact to client applications of any changes to the physical storage design or implementation.

Within the prototype, the OSD is provides a set of Data Access application program interfaces (APIs) to support the above goals. The Data Access APIs

provide application-level access (e.g., search, create, read, update, delete – SCRUD) to data entities stored in the System's underlying persistence solution (e.g., relational database).

4.3.4.2.1 Data Access APIs

Two approaches have been investigated to implement the Data Access APIs as part of prototyping:

1. The preferred approach entails implementing the DAO functions as network services rather than direct API calls. Under this option, data access services are defined in a particular language (e.g., Java), which receives DAO requests via network interfaces and responds with the retrieved data entities (RESTful interfaces[2]). A RESTful API is an API that uses HTTP requests to GET, PUT, POST and DELETE data. REST is an acronym for Representational State Transfer, which is an architectural style used for web development. This is a common approach in modern software systems.

   The primary advantage of this approach is that it uses ubiquitous technologies while minimizing the effort required for clients implemented with different languages to access persistent data. The potential disadvantage of this approach is the performance overhead (latency) introduced by data serialization and network transport, relative to direct API access. Based on its advantages, this approach has been selected for implementation in the Executable Architecture prototype pending further evaluation to establish the sufficiency of latency performance.

2. A second approach under consideration should the performance of option 1 above prove insufficient is to develop direct DAO library APIs in each of the languages selected for the System. Two variants of this approach have been considered:

   a. Implement the full set of access methods (SCRUD) using separate solutions available in each language for the selected data storage solution. This approach has the distinct disadvantage of requiring significant effort to develop a separate set of access APIs in each language added to the System. Another issue is the inherent difficulty in maintaining a consistent set of interfaces across languages with different features and libraries supporting access to a particular storage solution.

[2] https://en.wikipedia.org/wiki/Representational_state_transfer

b. A possibly more efficient approach entails developing DAO functions using C or C++, and using a toolset such as SWIG to generate bindings in the other languages of the System, much like the approach described above for the data model. As discussed for the data model, this approach requires the DAO software to be implemented in C or C++, which are lower-productivity languages for this purpose relative to Java and Python, and which likely are at greater risk for obsolescence in the next 20 years of the modernized Systems lifecycle.

Ultimately, the System may require a combination of the above options; the service-based approach should be used in cases where the network latency overhead is acceptable, and the library APIs (implemented manually or via SWIG interfaces) could be provided as an optimization in other cases. As mentioned above, the network service model will be developed as part of the Executable architecture until and unless an insurmountable performance limitation is established.

4.3.4.2.2 OSD Data Access Implementation Patterns

Implementing the OSD get, store, and subscribe operations in a RESTful data access layer supports ubiquitous use from a variety of programming languages while abstracting the selected database technology. Interface implementations may directly query databases, access relational databases using object relational mapping (ORM) libraries, query and combine results from multiple databases in a polyglot solution, etc.

The REST layer allows for web service based data retrieval and persistence using standard HTTP verbs such as GET, POST, PUT, PATCH, and DELETE against OSD URIs. HTTP requests contain claim check information. Depending on the request this information is provided either in the request body or in the URI. When the OSD provides data over REST it encodes the requested information in the REST response body. Using a standard data encoding such as JSON further isolates client applications from the database implementation. Binary encodings could be used for some data types (e.g. waveforms) if necessary to meet performance requirements.

OSD notifications to clients require server initiated communication. Though using web based technologies are based on client initiated communication, these notifications can still be achieved in several ways: the client can regularly poll a REST endpoint provided by the OSD mechanism to check for notifications; the client can expose REST endpoints used by the OSD to post notifications; the OSD can notify clients over a parallel communications channel such as WebSockets.

The OSD Mechanism's REST layer can be implemented using two primary patterns: custom defined endpoints and endpoints following Hypermedia as the

Engine of Application State (HATEOAS) principles. Custom defined endpoints have the advantage of allowing fine grained control of the information provided by each OSD query. Queries can be created to provide the exact set of information required by a particular client. The tradeoffs are design complexity and the potential proliferation of single use endpoints. Mitigating these concerns requires implementation diligence. Alternately, the HATEOAS principle can drive OSD endpoint implementations:

Benefits of the HATEOAS approach include strong de-coupling of the server and client (e.g. the client does not need to know the precise URIs underlying the links provided by HAL resources, allowing the underlying service calls and URIs to evolve independent of the client implementation). In addition, HATEOAS allows clients to follow links and discover information with little prior knowledge of the underlying model. Negatives of this approach include more service calls, possibly chains of service calls to follow a series of links in order to retrieve the desired resource. In a high-latency environment this may have significant performance impacts.

4.3.4.2.3 Data Storage of Waveforms

A key function of the OSD mechanism is to provide storage and retrieval of waveform data accumulated from the System's networks of stations. Waveform data sets differ from other information managed by the OSD in a few important respects:

- Waveform data sets are much larger than other information managed within the System, and account for most of the persistent data storage volume.
- As time series data, waveforms are managed as contiguous time blocks of unstructured binary data. Query-able metadata about waveforms is typically managed separately order to support queries.

These characteristics – significant volume and largely unstructured content – indicate potential advantages of a separate storage approach for waveforms. Whereas other System data are well suited to traditional database applications providing full Create/Read/Update/Delete access with sophisticated query support (e.g. relational database management systems), an alternative storage approach for waveform data is likely to improve efficiency & performance, as well as to simplify the overall OSD design.

The current US NDC System manages waveform data as collections of binary flat files stored on shared data volumes. Metadata describing the waveforms (e.g. network, station, channel, time range, etc.) are stored in accompanying RDBMS

records. Synchronization between the primary and backup sites is accomplished through parallel network distribution of incoming waveform data to both sites. There are a few important disadvantages with this approach for the modernized System:

- The modernized System architecture features a horizontal scalability model based on clustered deployment intended to address future scaling and availability needs. Management of the current file system approach in a clustered environment will likely introduce significant access latency (e.g. using NFS) and/or cost (e.g. using a Storage Area Network).

- The current approach requires custom application software to manage waveform time block files and provide retrieval of arbitrary time ranges across files. As discussed further below, there are a number of COTS time series storage technologies providing equivalent capabilities out of the box.

4.3.4.2.3.1 Prototyping

As part of an ongoing design effort, the prototyping team has investigated a number of waveform storage options, including the following:

- Relational Database

- Key/Value Stores

- Time Series Data Stores

The team completed a cursory assessment of a relational database solution, and as a result of the disadvantages discussed below, quickly shifted focus to an investigation and prototype of key/value stores. Having completed an initial prototype, the team is currently investigating time series-focused data store COTS solutions, which appear to have a number of advantages over key/value stores for waveform storage.

4.3.4.2.3.1.1 RELATIONAL DATABASE

The primary benefit of this approach is that it enables consolidation of waveform and other system data into a single storage solution, simplifying the technology stack. However, as discussed earlier, this likely is not optimal, given the unique characteristics of waveform data (large volume of unstructured binary data). Beyond the fact that relational databases offer little advantage for storing waveform data, the team's investigation identified a number of concerns associated with this approach, including:

- Read/write access is generally slower than alternatives. For example, access latency is necessarily higher relative to direct file system access for data on disk, and requires more tuning effort to achieve performance similar to key/value caches for data both in memory and on disk.

- Backups are generally more costly and complicated for large BLOB data sets in a relational database than for binary data stored in flat files on disk directly, or distributed in a non-relational system such as a distributed key/value store.

4.3.4.2.3.1.2 KEY VALUE STORES

In order to investigate key-value stores as an alternative for waveform storage, the team directed a student intern summer project that surveyed candidate solutions and ultimately developed an initial exploratory prototype using the Riak KV store. The initial phase of the project briefly investigated several open source solutions, including Riak KV, Apache Cassandra, CouchDB and Redis. Based on its feature set as well as the level of project maturity & prevalence, documentation quality and ease of provisioning & management, Riak KV was chosen as an exemplar for prototyping purposes. Key findings include the following:

- CouchDB utilizes a JSON-like storage format for keyed values, and thus is optimized for structured and semi-structured values that can be represented as documents. This format is not well suited for storage of largely unstructured waveform data.

- Redis historically was designed as an in-memory key-value store, and its support for persistent distribution/replication of data in a clustered environment was less well established than other solutions at the time of the investigation.

- Cassandra was found to be relatively complex to deploy and manage in previous investigative prototyping work. To avoid associated impacts on prototyping productivity, the team opted to use Riak KV as a more accessible solution.

- Riak appears to provide the desired efficiency, horizontal scalability & high availability features (cluster-based deployment with replication, automated failure discovery and failover, in-memory caching, etc.). The team found it straightforward to work with, and relatively simple to deploy & manage in a clustered environment.

In the second phase of the project, the team developed an exploratory software prototype using Riak KV deployed on a five-node VM cluster provisioned using

OpenStack. The Riak cluster was configured with recommended settings regarding definition of the key space, replication policies, etc. The team developed tests measuring read and write latency across parameter spaces - primarily record size, number of records, number of keys - using generated dummy data. This initial prototyping did not uncover any significant concerns with the technology, and in general the team was encouraged by the modest learning curve, deployment/management complexity and available APIs. The primary limitation of key value data stores as a general approach is that they do not provide built-in support for time-based operations such as retrieval of a time ranges crossing stored time blocks. This key capability requires custom software to be developed around the KV store APIs.

4.3.4.2.3.1.3 TIME SERIES DATA STORES

The team recently identified a number of time series-focused data storage solutions that will be considered in follow-on work. In principal, this technology seems optimally suited to the problem space, given the time-ordered nature of waveform data. An obvious advantage these technologies offer is support for time-based operations, including time range queries. Open source projects of interest in this category include: InfluxDb, Graphite, Druid, Riak TS, etc.

4.3.4.2.3.2 Follow-on Work

For the problem space of waveform storage, the immediate focus of the team moving forward is to evaluate time series database options. Based on the outcome of that investigation, the team will propose a design solution based on either key-value or time series data storage technologies as part of the architecture runway to support early product increments.

4.3.4.3 Processing Sequences

As described in Section 4.2.6.2, the System will provide the capability to execute pre-defined sequences of automated processing via the Processing Sequence Control mechanism (PSC). Key design goals driving the implementation of this capability include the following:

1. The System should provide a notation and toolset enabling the definition and deployment of Processing Sequences, including complex orchestration logic – e.g., parallel (fork/join) and repeated processing flows (i.e., loops), conditional flow execution, and nested sequence execution (i.e., one sequence invoked from within another).

2. The Processing Sequence Control mechanism should be capable of delegating processing defined in sequences to external applications and plugins implemented in the languages approved for the System.

In order to achieve the first goal, the prototyping team considered a number of alternative designs:

1. Distributed orchestration of processing components through runtime configuration of communication pathways between components based on a declarative (e.g., XML) syntax – under this approach, a sequence would be defined by linking the output channels (e.g., AMQP queues) of components to the input channels of other components using channel identifiers. Note that an approach similar to this one is currently used by the IDC system. The primary disadvantage of this approach is the difficulty of configuring complex orchestration logic such as loops and nested sequence execution via input/output coupling. For this reason, the approach was not included in the prototypes.

2. Centralized orchestration using an open-source COTS execution engine capable of directing components based on pre-defined processing sequence definitions. Various open-source technologies were evaluated for this approach, including Spring Batch and several Business Process Modeling and Notation (BPMN) tools. Based on the evaluation, the *Activiti* open-source BPMN engine was selected for use in the prototype. Activiti is an open-source Java runtime engine that executes processing sequences defined using the BMPN 2.0 standard notation. Advantages of Activiti include the following:

    - All of the orchestration constructs described in design goal 1 (fork/join, loops, conditional execution, nested sequences, etc.) - as well as others - are supported.

    - Processing sequences may be defined using either the BPMN 2.0 visual modeling notation (via an Eclipse plugin) or the XML notation.

    - The engine maintains the execution state of each processing sequence in a database, and supports transactions within processing sequences (e.g., including rollback upon failure).

For more information on the Activiti-based PSC design, see 5688: The System shall provide the System User the capability to remotely access required user interface functions on the OPS Subsystem from a remote connection over a secure connection.

5689: The System shall provide access to all Analyst capabilities from a remote location over a secure connection.

5690: The System shall be comprised of discrete subsystems each configured to support its mission, including: 1) Operational (OPS) Subsystem; 2) Alternate (ALT) Operational Subsystem; 3) Testbed (SUS/TST) Subsystem; 4) Development (DEV) Subsystem; 5) Continuous

5731: The ALT Subsystem shall be a copy of the OPS Subsystem in software and hardware not physically collocated with OPS.

5738: The System shall reuse suitable existing software where practical.

5739: The System shall use open-source software whenever possible.

5740: The System shall use open-source software when both open-source and commercial software are available.

5766: The System shall support at least 1000 Authorized External Users.

5767: The System shall support each Authorized External User requesting up to 4GB of data per day.

5768: The System shall support at least 30000 requests for data and products per day.

5831: The System shall use relational database management systems that support ACID transactions, referential integrity and fine grained locking.

5832: The System shall use a distributable open source database for Standalone Subsystems.

6442: The Standalone Subsystem software distribution shall be available for use by any authorized party without export restrictions.

Appendix B. BPMN For Processing Sequence Control.

3. Centralized orchestration via executable scripts defining processing sequences in a language such as Python. This approach replaces the declarative processing sequence notation and execution engine (such as Activiti) with an executable script. The primary advantage of this approach is that it provides a simpler, less complex solution (relative to e.g., BPMN engines) supporting all of the orchestration constructs identified in design goal 1. The disadvantages of this approach include the following:

    - Processing sequences are defined imperatively rather than declaratively, and there is no visual modeling notation.

    - Execution state tracking and transaction support are not included by default and would introduce significant complexity to add.

In order to achieve the second design goal above (cross-language support), the Executable Architecture prototypes focused on network-based communication between the PSC and Control Class applications (e.g., signal detectors) running in separate processes. The two technology pairs discussed in section 4.2.6.4 Inter-Process Communication (HTTP/JSON and AMQP/Thrift) were implemented to provide language-agnostic messaging and data encoding as part of the prototypes. In each case the PSC delegates processing to external applications by sending a request message with encoded data reference, configuration, and parameters. The external application receives the message, performs the requested processing, and responds with encoded result data and status.

Recall the primary limitation of HTTP/JSON approach is that HTTP is not well suited to patterns other than request/response (e.g., publish/subscribe). This is not truly a limitation given a centralized approach to the PSC mechanism where a central orchestrator (Activiti) delegates processing via request/response. However, under an alternative decentralized PSC design (e.g., where orchestration is achieved via configurable queues directly connecting applications), HTTP is less well suited.

The centralized PSC design implemented in the prototype (Activiti) uses request/response exclusively for communication between the PSC and target applications. For this reason and due to its relative simplicity and flexibility, the HTTP/JSON technology pair has been selected as the preferred approach for PSC/application communication within the Executable Architecture prototype. AMQP/Thrift may be considered as an alternative in future prototyping work should the comparative efficiency of this approach become compelling.

4.3.4.4 User Interface Frameworks

Another critical area of investigation for Executable Architecture Prototyping is evaluating options for the implementation of the System user interface. Prototyping activities have evaluated three primary options for the implementation of the System user interface:

1. Rich-client Java application
2. Browser-based web application
3. Natively-deployed web application

Early prototyping activities explored the development of a rich-client Java application using the NetBeans Rich Client Platform (RCP). This prototype heavily leveraged capabilities provided by NetBeans, including its modular plugin framework and support for user-customizable display layout and data synchronization across displays. However, UI design options are somewhat limited by standard Java look and feel. Moving beyond those design options would require intensive custom rendering, which would be expensive.

More recent prototyping activities explored development of the System user interface using web technologies (e.g., JavaScript, HTML, CSS) deployed via a standard web browser. Benefits of a web-based UI include more control and freedom over the UI design and the ability to leverage the momentum of web development in industry by making use of open-source JavaScript frameworks that support UI development. The key additional benefit of a browser-based web UI is a simplified deployment and upgrade model, which is likely more advantageous for some use-cases (e.g., support for Authorized External Users) than for others (e.g., core analysis capabilities).

A third UI implementation option is a natively-deployed web application, which implements the UI using the same web technologies mentioned above but deploys the application natively to dedicated workstations instead of via a browser. Prototyping activities have explored the use of Electron, a framework for deploying cross-platform native applications using web technologies. The benefit of this hybrid approach is the ability to leverage modern web technologies while maintaining the benefits of a desktop application, including access to the filesystem and native menus and notification mechanisms. However, this approach requires a more traditional installation and upgrade model.

With all three UI implementation options, the proposed back-end System architecture remains the same. The UI accesses Control classes via RESTful web service calls to a server back-end. Data synchronization occurs via WebSocket technology, a web standard allowing for full-duplex communication between the

server and UI client. Using WebSockets, data is pushed to the UI, which then updates itself accordingly. In all cases, web-services could be forward-deployed to the workstation, as needed, to improve System performance. Thus, selecting the best UI implementation approach is not influenced by the back-end architecture, but rather, is based entirely on the benefits of one UI-specific technology stack over another.

Key criteria for selecting the UI implementation approach include:

- **Interactive performance**: Any viable UI technology stack must provide acceptable performance for Analysts under both typical and stressing conditions. Since the back-end implementation of web services is the same for all proposed UI implementation options, this criterion is focused strictly on performance in the UI itself. Examples of performance assessments include the ability to load and interact with large amounts of waveform data and the ability to synchronize data across multiple complex displays (e.g., an event list, waveform display, map display, etc.) without degrading performance for key, time-critical operations. Prototyping activities have explored the feasibility of holding large amounts of data in-memory in the browser. In some cases, waveform segments can be loaded on-demand from the server; however, in many cases, the UI will need to store a large number of waveforms in memory. Using a 64-bit build of modern browsers, initial prototyping activities found no performance degradation when dealing with large heap size, with the following caveats: Chrome restricts each tab to 4GB of memory for security reasons; however, this limit can be increased using runtime flags. Chrome also restricts GPU-render processes to a larger 16GB limit, which is where the WebGL-powered waveform display would reside. The Electron framework, which is built on Chromium, behaves similar to Chrome in that it requires runtime flags to increase heap size. Firefox does not appear to have the same heap size restrictions. Thus far, these performance assessments indicate that UI web technologies can provide a user experience that is equally or more performant that a rich-client Java application.

- **Saving and restoring user-defined workspaces:** A key requirement of the System is that end-users have the ability to organize their displays into custom workspaces that can be saved and restored across user sessions. This requirement is well-supported by the NetBeans RCP, as demonstrated by early prototyping activities. In terms of web-based alternatives, we have explored Golden Layout, a third-party JavaScript framework that provides similar layout management capabilities within the confines of a browser. Golden Layout supports opening, closing, dragging, dropping, and resizing individual tabbed displays within a single browser window. Additionally, tabbed displays can be undocked

into a separate browser window, which maintains communication with the parent window. This technology is very promising for supporting layout management requirements; however, it does have limitations. For instance, the ability to save and restore the placement of displays across multiple screens is limited by current browser technology. This risk could likely be mitigated by using Golden Layout in a natively-deployed application (e.g., Electron) rather than deploying via the browser. Regardless of the deployment model, Golden Layout is the leading candidate for web-based layout management support; however, the code base will require significant refactoring to fully support our robust operational use-case.

- **Integration of GIS capabilities:** A goal of the modernization effort is to provide robust GIS capabilities that are well-integrated with the rest of the application. In recent years, many industry leaders in GIS (e.g., Google, ESRI, AGI) have moved decidedly in the direction of web-based tools, either instead of or in addition to their traditional thick-client offerings. The web-based GIS tools available today are improving rapidly. Although they are currently less mature than their thick-client counterparts, this is likely to change as demand increases and industry leaders continue to devote resources to their web offerings. A web-based GIS technology stack might be implemented using a combination of open source tools (e.g., GeoServer, Cesium, OpenLayers, Turf.js) or perhaps via the ArcGIS for Server platform. For a rich-client Java implementation, NASA World Wind is a likely candidate for map-based visualization. Regardless of the chosen technology stack, GIS capabilities will be implemented based on OGC standards (e.g., Web Feature Service, Web Map Service, Web Coverage Service).

- **Ease of deployment and upgrades:** As mentioned previously, accessibility and ease of deployment are key benefits of a browser-based web UI. The ability to access the System user interface via a standard web browser means that the application doesn't need to be installed on individual workstations and that upgrades can be deployed once to a server and made available immediately to all users. However, this model is limited to a browser-based deployment, and the benefits would be diminished for users without network connections (e.g., on a standalone system) and in cases where web services need to be forward-deployed (and thus installed on individual workstations) to meet performance needs.

- **Scalability, complexity, and maintainability of the code base:** This criterion combines a number of factors related to the overall complexity of the UI code base. This includes the learnability of languages and technologies, the availability of resources and an active community of

developers to learn from, and the availability of robust development and testing tools. From a code complexity standpoint, a thick-client Java implementation is less risky due to its object-oriented nature and well-established patterns. As mentioned previously, a key benefit of web technologies is the ability to leverage the momentum of literally hundreds of available third-party JavaScript libraries and frameworks. However, this is also a risk of web technologies – complexity of the code base increases dramatically with the need to integrate a large number of disparate third-party libraries into one coherent and maintainable code base. The key to mitigating this risk is to accept the reality that some of these third-party libraries will need to be replaced during the multi-year development cycle of the application. If the UI architecture is designed with this eventuality in mind, then the code can be organized so that UI libraries can be replaced more easily without negatively impacting the rest of the UI code or the application logic layer. JavaScript, as a dynamic, untyped language, adds additional concern in terms of the complexity, scalability, and maintainability of large-scale code bases. Prototyping activities have begun to mitigate this risk by making use of Typescript, a typed superset of JavaScript developed by Microsoft, which is designed to make the management of large-scale JavaScript applications easier.

UI prototyping activities have focused on working enough with JavaScript and available third-party libraries to make an informed decision about the performance, interaction model, and complexity of code for a large-scale system. To date, the key third-party libraries that we have used for UI prototyping include:

- React: Supports efficient update and rendering of UI components based on state. Developed by Facebook and used widely in industry. A likely candidate for the foundational framework of the System UI. (https://facebook.github.io/react/)

- Golden Layout: A multi-window layout manager for web applications. Supports RCP-like look and feel inside the browser; however, the library will require rework to fully support our robust use-case. (https://www.golden-layout.com/)

- Cesium: An open-source 2D/3D virtual globe implemented with WebGL. Developed by AGI. Leading candidate for web-based map visualization because it is one of only a few options that currently support 3D rendering. (https://cesiumjs.org/)

- ag-Grid: A data grid library that provides robust capabilities for the display of tabular data. Provided an initial test case for wrapping third-party tools for easy replacement in the future. (https://www.ag-grid.com)

- Redux: A library for managing complex application state. Used for initial investigation of undo/redo capabilities in the UI. (http://redux.js.org)

- Electron: A framework for creating native applications using web technologies. A likely candidate for deployment for scenarios where browser-based deployment is not viable. (http://electron.atom.io/)

- Typescript: A typed superset of JavaScript developed by Microsoft. A likely candidate for use since it has already made our relatively small UI prototyping code base easier to manage. (https://www.typescriptlang.org/)

Based on the criteria defined above, and on what we have learned through prototyping activities thus far, we believe that the benefits of UI web technologies outweigh the risks. Thus, the proposed path forward is to develop the System user interface using web technologies, meaning that the core programming languages for UI development will be a combination of JavaScript, HTML, and CSS. Third-party libraries and frameworks will be leveraged whenever possible to support the development of robust and consistent displays and also to support the maintainability of the code base.

Ongoing prototyping activities will continue to buy down risk associated with a browser-based deployment (e.g., by resolving issues associating with user-defined workspaces and by continuing to explore performance considerations, GIS capabilities, and tools to improve code maintainability and developer efficiency). If a browser-based deployment proves inviable for some use-cases (e.g., core analyst capabilities), then we will continue to develop a web-based UI but deploy it natively to individual workstations. This approach allows prototyping activities to move forward with the evaluation and selection of specific web technologies, since the same technologies will be used regardless of the eventual deployment model. Additionally, it is likely that the same code base could be deployed both natively and via the browser under different usage models without significant extra development effort.

4.3.4.5 Undo - Redo Implementation

Undo-redo provides the means for Analysts to quickly recover the previous state of an Event or a Signal Detection. Associated with web-based user interface prototyping, the prototyping team investigated the undo-redo capabilities inherent in the Redux third-party library. Prototyping used Redux to capture and distribute the state of display variables to synchronize multiple distinct displays. Redux also offers some capability to save previous display state by storing and manipulating snapshots of state in memory. Redux maintains an underlying immutable state structure by creating a new state object for each state transition. The performance impact of this approach is low as most of the new state simply contains references to previous state objects, except for the

modified sections. A limitation of this approach is the state structure is confined to the user interface and any state transitions must be duplicated on the server side. Further work will focus on an approach to maintain consistency between the user interface and the server side while providing required user interface performance.

4.3.4.6 Provenance Implementation

Requirements and use cases indicate the System must capture, index, store, and query provenance information describing which results were created in the System, when they were created, and how they were created. Developing a useful provenance framework requires anticipating future uses of the provenance information in order to capture and store the information necessary to respond to those queries. The architecture team has considered two primary approaches for capturing and storing provenance information:

1. Event sourcing
2. Relational database schemas

The prototyping team developed an Executable Architecture prototype to evaluate the event sourcing[3] pattern. Storing provenance in predefined relational database schemas was not prototyped as the team has extensive experience developing applications which interact with relational databases.

Event sourcing is a pattern where a series of domain events record all changes to object state and where the object state can be recreated from those domain events. The pattern allows recreation of an object's state at any point in its history without having to explicitly store each of those states.

Initial prototyping activities surveyed the available event sourcing libraries. The selection of Java as the primary implementation language limits the viable options to:

- Axon Framework: Axon is a Java distributed programming framework providing a programming model based on Command Query Responsibility Segregation (CQRS[4]) and event processing patterns. Event processing, messaging, and storage are core concepts in Axon. Axon's APIs follow Domain Driven Design nomenclature so they fit well with theoretical discussions of event sourcing found in articles and books. Axon supports a variety of database technologies including PostgreSQL and MongoDB.

[3] http://martinfowler.com/eaaDev/EventSourcing.html
[4] http://martinfowler.com/bliki/CQRS.html

- Akka Toolkit: Akka is a distributed programming framework based on the actor model. Akka supports event sourcing using persistent actors. Each actor is responsible for creating and storing events corresponding to the results of commands issued to the actor. Event sourcing is a means of persisting actors but is not fundamental to the Akka programming model. As such, Akka was eliminated from further Executable Architecture prototyping.

The Axon event sourcing prototype supports the following:

- Experience implementing the CQRS and Event Sourcing patterns: Increasing team familiarity with the concepts and complexity introduced by the CQRS and Event Sourcing patterns, including using separate data representations for command handling and query objects, exclusively using domain events to update aggregate objects in response to commands, and communicating state changes via domain events.

- Axon performance evaluation: evaluating the performance of Axon's command handling, event handling, event sourcing, and CQRS implementation in a distributed environment. This test uses a Python script sending requests to REST controllers to study Axon's command handling throughput for both breadth (many event sourced Aggregates) and depth (event sourced Aggregates with many corresponding domain events) of domain event histories. The performance test is subjective since it is not compared with a baseline performance metric.

- Build/Adopt framework decision: determine if the Axon framework is suitable for use in System development. Since Axon is the only feasible framework discovered during the event sourcing library survey the alternative to using Axon is develop a custom CQRS and event sourcing framework. The custom framework could either be based on Axon or developed from scratch.

Figure 4-9 shows the structure of the Axon event sourcing and CQRS prototype. The prototype uses the most recent release of Axon, version 2.4.4. The prototype runs on a 5 node OpenStack cluster. The user interface is a Python script that makes requests to REST endpoints using JSON over HTTP. The REST controllers use the requests to create Command objects and publish them to a command bus which routes Commands to the appropriate Command Handler. Axon provides horizontally scaled command handling using JGroups clustering to distribute the command bus. Command Handlers invoke operations in the Aggregate classes to run algorithms, update values, etc. Aggregates for Event, Event Hypothesis, Signal Detection, and Signal Detection Hypothesis exist in the prototype. Each Aggregate is event sourced so Domain Events record changes to their state. The Domain Events are stored in the MongoDB event store and are published to the Event Bus for delivery to Event Handlers via RabbitMQ

messaging. Axon also horizontally scales event handling. Axon implements these clusters using RabbitMQ messaging. The Repository is an abstracted data store for the Aggregates. The Repository is responsible for managing the Domain Event to Aggregate mapping and can be used to recreate Aggregate state by replaying Domain Events.

In CQRS the portion of the System responsible for handling Commands is separate from the portion of the System responsible for responding to queries. The top portion of Figure 4-9 is responsible for receiving, processing, and ultimately storing the results of commands (i.e. Domain Events) in the MongoDB Event Store. However, since the Event Store provides minimal query power a separate PostgreSQL database maintains a separate view of the information in a format better structured for querying. This is the query response side of CQRS. The prototype uses domain event handlers to update PostgreSQL as domain events are published to the event bus. Applications such as GUIs and Control classes can query the database for the results of command processing.

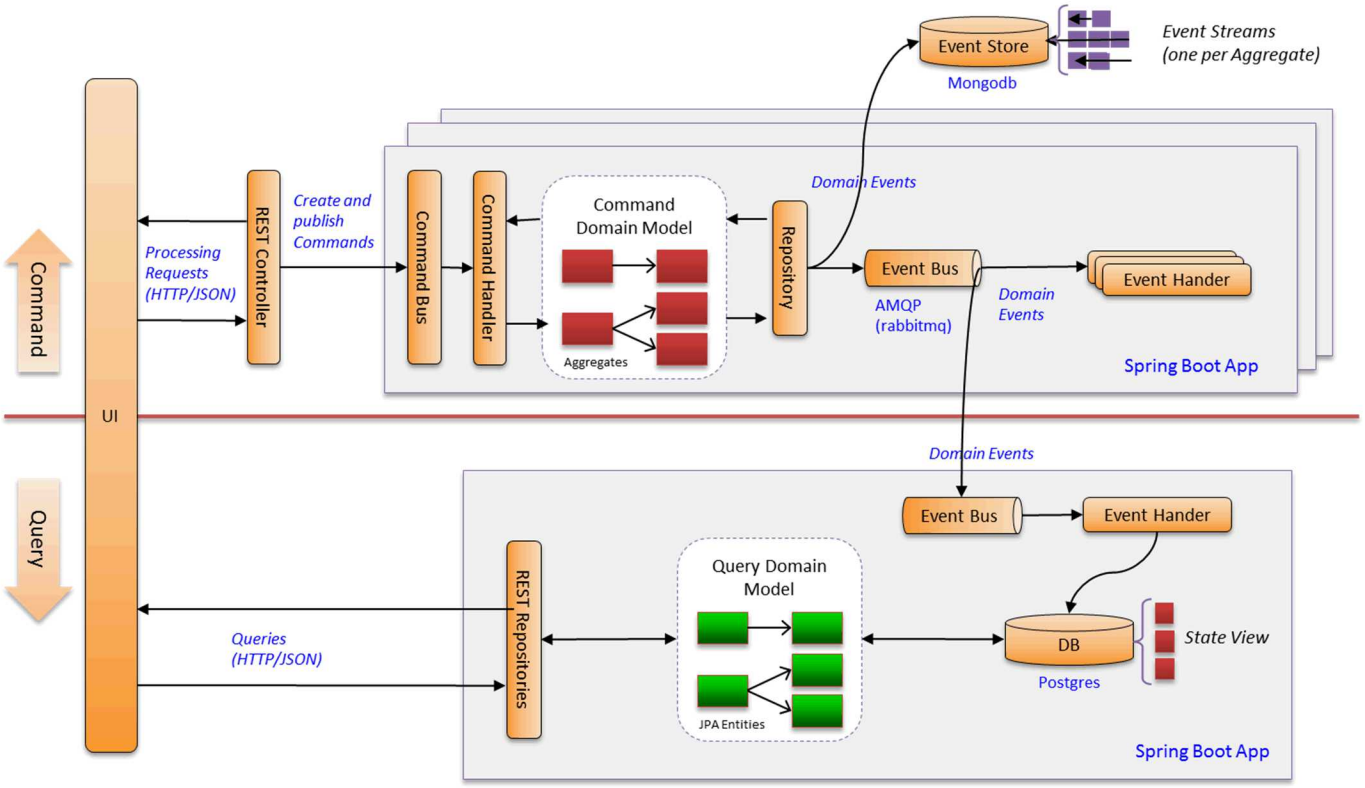

Figure 4-9. Implementation View Axon framework event sourcing prototype

Overall the prototyping team found Axon 2.4.4 to provide a flexible and understandable API. Key framework components such as the domain event store, messaging system, and Aggregate storage have extensible implementations based on Java interfaces. Axon provides default implementations using common technologies such as relational storage via JPA,

document storage via MongoDB, and AMQP messaging via RabbitMQ. In the event these technologies need to be replaced with alternate implementations the team has confidence the changes can be made with minimal impact to the remainder of the Axon framework. Additionally, since Axon defines annotations allowing it to be used in Spring Boot applications it integrated easily in the prototyping environment.

The prototyping team has several concerns with the Axon framework:

- JGroups distributed command bus: Axon uses JGroups to distribute the command bus and AMQP to distribute the event bus. Axon's choice to use JGroups for the command bus introduced dependence on an additional third party library. While stable, JGroups is an older technology. If it becomes necessary to replace JGroups possible alternatives include implementing a new distributed command bus using a different technology (e.g. AMQP messaging) or implementing load balancing at the REST endpoints (e.g. using NGINX) rather than within Axon.

- Limited development team: Reviewing Axon's GitHub repository reveals it is primarily developed by one person with significant efforts from one or two other contributors. The implication is long term maintenance and support of Axon may be limited. If long term support is unavailable the development team will have to maintain or replace Axon.

- Evolving API: The prototype uses the current release of Axon, version 2.4.4, but this version is anticipated to be obsolete in the near future when Axon 3.0 is released. This release is known to have some architectural differences from Axon 2.4.4. While the team's experience and understanding of CQRS and event sourcing in Axon will remain valuable, the move to Axon 3.0 will require the team to become familiar with new APIs when it is released.

- Documentation and support: Axon has some documentation online but much of it is high level enough that it provides more theoretical than practical information. Axon has some support via Google groups. While timely, help is generally provided by the primary Axon developer.

After evaluating the Event Sourcing prototyping results, the Architecture Team selected to store provenance information along with the processing results embedded in the relational database schema. This approach requires identification of the provenance information at development time. While the Axon implementation proved well designed and supported adequate performance, the team determined that the additional complexity of separate storage of processing results from provenance information, multiple storage solutions, and replaying domain events to retrieve current state was a risk for developing and maintaining the System. Additionally, the team decided that the

identification of provenance information during implementation was adequate to meet requirements.

4.4 Process View

The Implementation View specifies how Analysis Classes are physically organized at development time to support development. The Process View, on the other hand, specifies how Analysis Classes (**Control** classes in particular) execute within processes at runtime to achieve concurrency in the system.

The System is fundamentally distributed. From the Process View perspective, the system is essentially a large set of cooperating processes and threads spread across several processing nodes. A distributed architecture such as this provides many advantages, including:

- The ability to take advantage of multiple CPUs within a node and across multiple nodes to perform activities in parallel.
- The ability to react rapidly to certain types of external stimuli, including time.
- Increased CPU utilization, by allocating the CPU to other activities while some thread of control is suspended waiting for some other activity to complete (e.g., access to some external device, or access to some other active object).
- The ability to better prioritize activities.
- The ability to better separate concerns between different areas of the software.
- Higher system availability.
- Increased scalability in order to meet future unknown requirements.
- Ability to monitor individual processes to evaluate system status and performance.

There are limitations in distributed architectures as well. In general, such systems are harder to design, build and test. The design is sensitive to inter-process communication and too much inter-process communication can hurt system performance. For a system with data processing requirements and several operators using the system in parallel however, the benefits outweigh the limitations.

4.4.1 Use of Multi-Threading

The System will take advantage of the underlying multi-processor platform through the use of multiple processes and multi-threading within processes. Future prototyping efforts will define how multi-threading will be implemented within the system.

4.4.2 Process View Mappings

The Process View defines the processes that collectively execute the system and how Analysis Classes are mapped to those processes. When considering the Process View, it is often useful to also consider the nodes on which the various processes execute—a focus of the Deployment View. Analysis class to Process mappings will be defined in future iterations.

4.5 Deployment View

4.5.1 Deployment Considerations

The initial assumption for the System is that all processing nodes will be x86-based processors running the Red Hat Enterprise Linux OS. The processing nodes will run on virtualized hardware. Virtualization is employed for several reasons, including:

- *Flexibility in node provisioning* – Resources such as CPU and memory can be reassigned among virtual nodes without changing the underlying hardware.
- *Simplification of hardware maintenance* – Virtual nodes can be reassigned among hardware units to allow maintenance on faulty units.
- *Simplification of application upgrades* – Image-based installation bundles of application and OS software can be created and installed together. Virtual nodes can be configured quickly to run different software releases.

The design of the system architecture seeks to minimize the dependencies between the application software and the underlying computer hardware to provide the capability to upgrade to newer hardware, as it becomes available. The System is long-lived therefore the capability to upgrade to later versions of hardware is highly desired.

The term compute resource has been used to mean any physical or virtual component of limited availability within the System, and includes but is not limited to the following:

- Nodes,
- Workstations,
- Servers, and
- Compute Devices.

Virtualization and container technology are being applied to the development process to ease development, testing, integration and delivery. Containers will be evaluated as a method to facilitate software deployment.

4.5.2 Subsystems

The System is comprised of several subsystems each configured to meet its assigned mission. The primary mission is to monitor compliance with existing and future nuclear weapons testing treaties. This mission is performed by one of two subsystems – each capable of supporting the primary mission – installed at geographically separate locations to provide survivability in disaster scenarios such as weather or power outages. The subsystem located at the primary operational location is called the Operational (OPS) Subsystem and the other subsystem is called the Alternate (ALT) Subsystem. Only one of these two subsystems holds the primary mission at any given time, while the other system serves as the backup. Remote access to either the OPS or ALT Subsystems will also be available from a set of workstations at an independent location to provide coverage by accessing either the OPS or ALT subsystem. The Training Subsystem provides analyst training to operations personnel. The Standalone Subsystem contains a subset of the primary subsystem features in functionality, but operates independently.

Additional Subsystems facilitate software and hardware maintenance. The Sustainment/Test (SUS/TST) Subsystem provides testing, verification, and validation for upgrades. The SUS/TST Subsystem along with the Sustainment Alternate (SALT) Subsystem, provides a testing environment for subsystem synchronization and the transfer of operations from the OPS to the ALT. The Development (DEV) Subsystem and the Continuous Automated Testing Subsystem (CATS) provide development and testing of software upgrades.

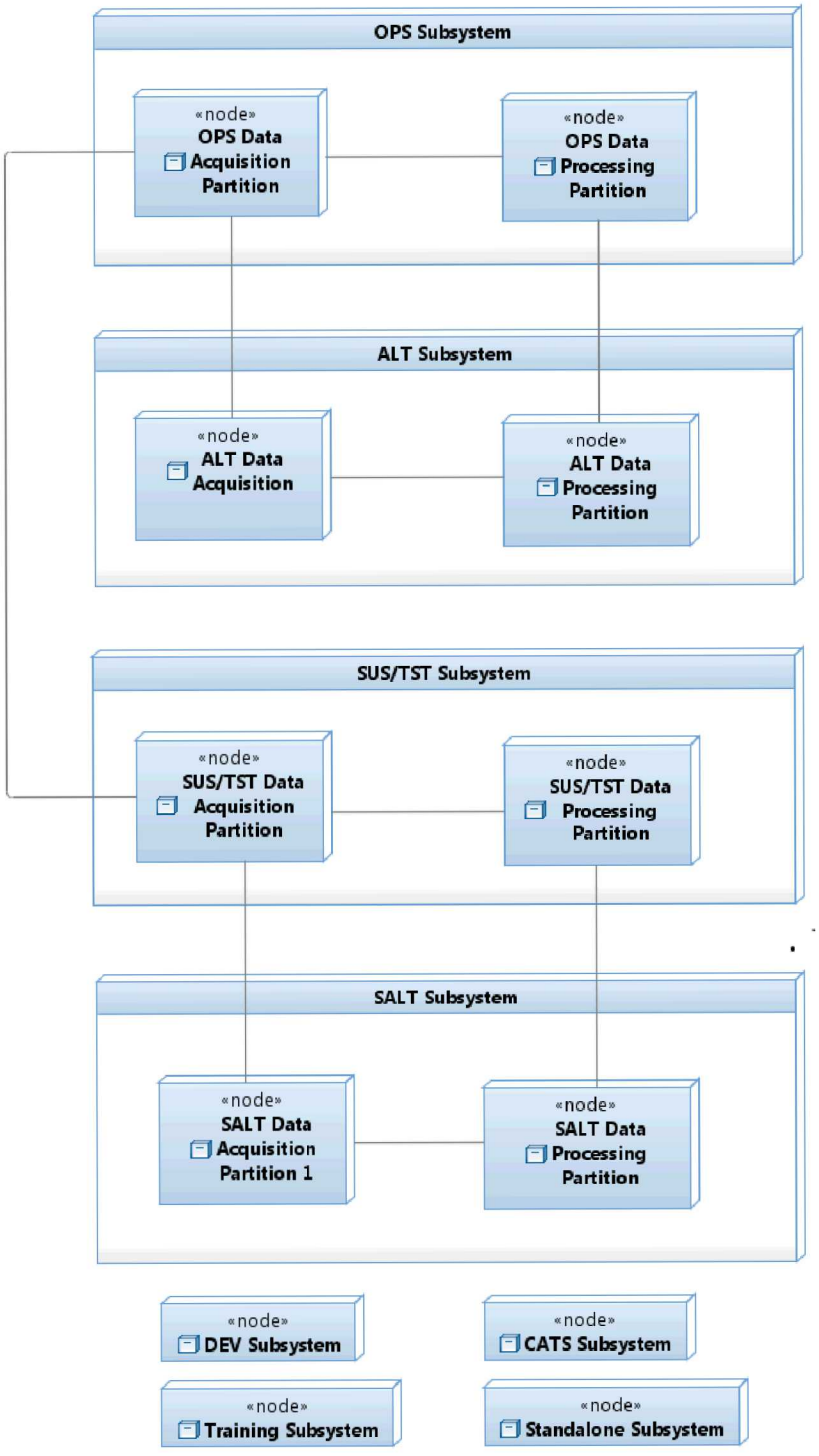

Figure 4-10. Deployment View

The deployment view resources needed to implement the functional requirements will be determined from a combination of experience with the prior developed and fielded system, analysis of the functional requirements in the System Specification Documents.

The compute resources will be logically decomposed into sets designed to support the functional requirements of each subsystem.

4.5.3 Device and Network Interfaces

This section describes device and network interfaces that have an impact on the system architecture. This section will describe significant devices and network interfaces in future iterations.

4.5.4 Deployment View Mapping

The Deployment View maps the processes defined in the Process View to the hardware processing nodes of the Deployment view. The Process-to-Node mappings will be defined in the Architecture Document in future iterations.

4.5.5 Procurement Considerations

The operational installations are responsible for the procurement requirements dealing with procurement procedures and issues which will be addressed on a site-by-site basis by personnel charged with those responsibilities.

4.5.6 Configuration Management

An integrated Configuration Management (CM) system shall be used to track the history of delivered items as well as the overall configuration of each system release. A configuration in this context includes items such as the System application (executables, libraries, and configuration files), the VMs including the installed operating system (OS) (VM provisioning, OS executables, libraries, and configuration files), network equipment configuration files, and associated documentation (on-line help). The full set of configuration items shall be documented in the System Configuration Management Plan.

5 APPENDIX A. SPECIFICATIONS

The following System specifications shall be satisfied by the System Architecture:

Common Specifications

2028: The System shall use a common object interface (data model and methods) for data.

2042: The System shall store automatic and interactive processing parameters in the database.

2043: The System shall store automatic and interactive processing results.

2139: The System shall report failures, warnings and notifications using a common messaging infrastructure.

2218: The System shall make use of commercial off-the-shelf (COTS) and open source software where possible.

2219: The System shall use commercial off-the-shelf (COTS) and open source software with a defined upgrade path.

2220: The System software shall be written using a minimum number of programming languages.

2224: The System shall implement dates and times that include leap years and seconds.

2226: The System shall use year 2038 epoch rollover compliant date formats.

2233: The System software shall be maintained and controlled via configuration management software.

2262: The SUS/TST Subsystem shall be a functionally redundant copy of the OPS Subsystem.

2317: The System shall maintain a mission profile operating 52 weeks a year, 7 days per week, and 24 hours a day.

2331: The System shall store on the System all existing data and five (5) additional years of data.

2332: The Training Subsystem shall provide storage with sufficient capacity to accommodate thirty (30) days of multi-phenomenology waveform data for stations available on the OPS Subsystem.

5703: The System shall provide the System User the capability to export the current view to a standard graphic format (e.g. TIFF, JPG or PNG)

5725: The System shall use date formats with four digit years.

IDC Only Specifications

5688: The System shall provide the System User the capability to remotely access required user interface functions on the OPS Subsystem from a remote connection over a secure connection.

5689: The System shall provide access to all Analyst capabilities from a remote location over a secure connection.

5690: The System shall be comprised of discrete subsystems each configured to support its mission, including: 1) Operational (OPS) Subsystem; 2) Alternate (ALT) Operational Subsystem; 3) Testbed (SUS/TST) Subsystem; 4) Development (DEV) Subsystem; 5) Continuous

5731: The ALT Subsystem shall be a copy of the OPS Subsystem in software and hardware not physically collocated with OPS.

5738: The System shall reuse suitable existing software where practical.

5739: The System shall use open-source software whenever possible.

5740: The System shall use open-source software when both open-source and commercial software are available.

5766: The System shall support at least 1000 Authorized External Users.

5767: The System shall support each Authorized External User requesting up to 4GB of data per day.

5768: The System shall support at least 30000 requests for data and products per day.

5831: The System shall use relational database management systems that support ACID transactions, referential integrity and fine grained locking.

5832: The System shall use a distributable open source database for Standalone Subsystems.

6442: The Standalone Subsystem software distribution shall be available for use by any authorized party without export restrictions.

6 APPENDIX B. BPMN FOR PROCESSING SEQUENCE CONTROL

A number of COTS or open-source solutions have been developed around BPMN, including tools (e.g., Eclipse plugins) for defining BPMN Processing Sequences as well as engines that support the execution of Processing Sequences based on their BPMN descriptions. Activiti is one of the more prominent open-source BPMN runtime solutions, with an active development community and strong user base.

The primary BPMN elements used to define Processing Sequences in the Executable Architecture prototype are described below.

1. *Activities* represent processing work to be completed as part of the sequence. Activities may be either atomic or compound, as represented in the sub-types below. Activities may be executed repeatedly (i.e., in a loop), conditionally (see Exclusive Gateway), in parallel (see Parallel Gateway), and at specific times or at specific intervals (see Timer-related elements).

    a. *Tasks* represent atomic processing actions within the sequence. A common task example is the invocation of a processing service (either within the local process space or via a network service interface).

    b. *Sub-Processes* represent nested processing sequences invoked from within an enclosing sequence. Sub-Processes are used to define compound processing flows (potentially including activities, gateways, sequence flows, events, etc.) within a processing sequence

    c. *Call Activities* are used to represent the invocation of one processing sequence from within another, where control is transferred to the invoked sequence during its execution.

2. *Sequence Flows* define the flow relationships between activities, gateways, events, etc. as well as the order of execution. Sequence flows represent the control flow, rather than flow of data between elements. The Processing Sequence Control mechanism supports the passing of data and configuration information between activities within a sequence; however, data interfaces are not part of the processing sequence definition. These interfaces are defined as part of the interface contract for services that implement the defined Activities.

3. *Gateways* are used to define conditional and parallel execution of Activities within a processing sequence as represented in the subtypes below.

    a. *Exclusive Gateways* define decision points within a processing sequence that control the execution of one or more alternative activities based on evaluation of a Boolean expression. Exclusive Gateways allow only a single sequence flow to be executed; the first sequence flow (in the order

defined) for which the gateway condition evaluates to true will be executed.

b. *Inclusive Gateways* also define decision points within a processing sequence that control the execution of one or more alternative activities based on evaluation of a Boolean expression. Unlike Exclusive Gateways, Inclusive Gateways allow multiple sequence flows to be executed; all sequence flows for which the gateway condition evaluates to true will be executed.

c. *Parallel Gateways* define parallel execution paths without conditions. All outgoing sequence flows from a parallel gateway are executed in parallel. Parallel Gateways are used to represent both the divergence and synchronizing convergence points within a processing sequence. All sequence flows incoming to a Parallel Gateway will be executed before the single outgoing flow is executed.

d. *Event-Based Gateways* define parallel execution paths that depend on the occurrence of events in the system (see the description of *Events* for more information.

4. *Events* represent occurrences within the system that affect the execution of a processing sequence. Example events include timer firings, receipt of messages, error notifications and other named signals. Events are typically used to control execution flow – i.e., where flows are executed only in response to one or more events.

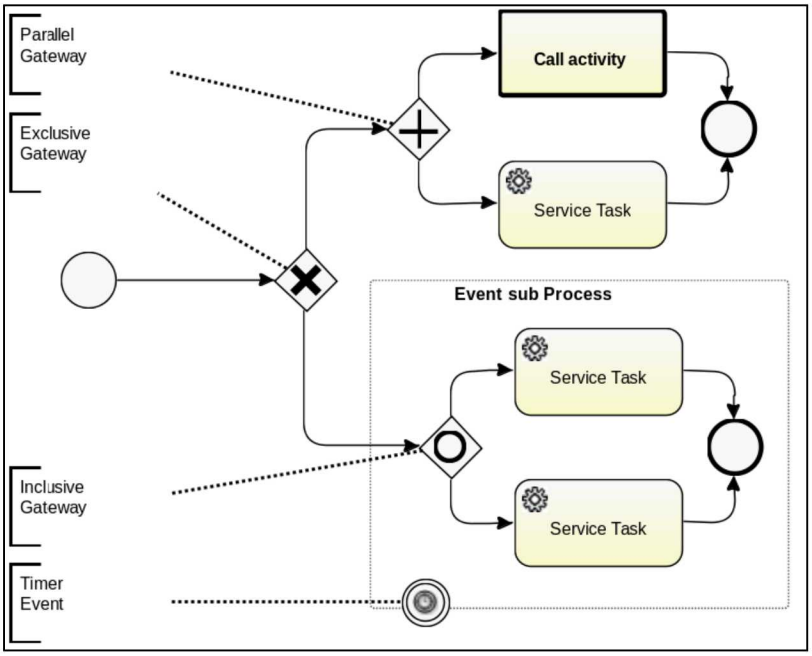

Figure 6-1. Example BPMN 2.0 Visual Modeling Notation

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
  <process id="myProcess" name="My process" isExecutable="true">
    <startEvent id="startevent1" name="Start"></startEvent>
    <exclusiveGateway id="exclusivegateway1" name="Exclusive Gateway"></exclusiveGateway>
    <serviceTask id="servicetask3" name="Service Task"></serviceTask>
    <callActivity id="callactivity1" name="Call activity"></callActivity>
    <subProcess id="eventsubprocess1" name="Event sub Process" triggeredByEvent="true">
      <inclusiveGateway id="inclusivegateway1" name="Inclusive Gateway"></inclusiveGateway>
      <serviceTask id="servicetask1" name="Service Task"></serviceTask>
      <serviceTask id="servicetask2" name="Service Task"></serviceTask>
      <sequenceFlow id="flow4" sourceRef="inclusivegateway1" targetRef="servicetask2"></sequenceFlow>
      <sequenceFlow id="flow5" sourceRef="inclusivegateway1" targetRef="servicetask1"></sequenceFlow>
      <endEvent id="endevent2" name="End"></endEvent>
      <sequenceFlow id="flow6" sourceRef="servicetask1" targetRef="endevent2"></sequenceFlow>
      <sequenceFlow id="flow7" sourceRef="servicetask2" targetRef="endevent2"></sequenceFlow>
      <association id="association3" sourceRef="inclusivegateway1" targetRef="textannotation3"></association>
    </subProcess>
    <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="exclusivegateway1"></sequenceFlow>
    <sequenceFlow id="flow3" sourceRef="exclusivegateway1" targetRef="inclusivegateway1"></sequenceFlow>
    <parallelGateway id="parallelgateway1" name="Parallel Gateway"></parallelGateway>
    <sequenceFlow id="flow8" sourceRef="exclusivegateway1" targetRef="parallelgateway1"></sequenceFlow>
    <sequenceFlow id="flow9" sourceRef="parallelgateway1" targetRef="callactivity1"></sequenceFlow>
    <sequenceFlow id="flow10" sourceRef="parallelgateway1" targetRef="servicetask3"></sequenceFlow>
    <boundaryEvent id="boundarytimer1" name="Timer" attachedToRef="eventsubprocess1" cancelActivity="true">
      <timerEventDefinition></timerEventDefinition>
    </boundaryEvent>
    <endEvent id="endevent3" name="End"></endEvent>
    <sequenceFlow id="flow11" sourceRef="callactivity1" targetRef="endevent3"></sequenceFlow>
    <sequenceFlow id="flow12" sourceRef="servicetask3" targetRef="endevent3"></sequenceFlow>
    <textAnnotation id="textannotation1">
      <text>Parallel
Gateway</text>
    </textAnnotation>
    <association id="association1" sourceRef="parallelgateway1" targetRef="textannotation1"></association>
    <textAnnotation id="textannotation2">
      <text>Exclusive
Gateway</text>
    </textAnnotation>
```

Figure 6-2. Example BPMN 2.0 XML Notation

The Processing Sequence Control mechanism executes Processing Sequences based on triggering events in the system. Example triggers include the following:

- **Timer events** – Processing Sequences may be executed at pre-configured times or intervals (e.g., periodically checking for new waveform data to process).

- **Service Invocation** – Processing Sequences may be executed based on invocation of the Processing Sequence Control mechanism's service interfaces (via API or RESTful webservice). This type of trigger supports execution based on operator commands and other events in the system; for example, for post-processing of created/modified data entities (signal detections, event hypotheses, events, etc.), processing stages, etc.

- **Data Subscription Callbacks** – The Processing Sequence Control mechanism maintains subscriptions for select data updates in the system that require a processing response (e.g., the creation of a new event). These subscriptions and the corresponding Processing Sequence(s) are installed as configuration items in the system. When the Processing Sequence Control mechanism receives callbacks for configured data subscriptions, it invokes the associated Processing Sequence(s).

The Activiti-based Processing Sequence Control mechanism supports a scalable, distributed processing model for execution of processing sequences. As depicted in Figure 6-3, tasks executed by the Processing Sequence Control mechanism may be

implemented as service invocations routed to control classes running in separate processes, potentially on separate hosts within the system. This approach allows for parallel execution of Activities within a Processing Sequence across multiple processes and nodes.

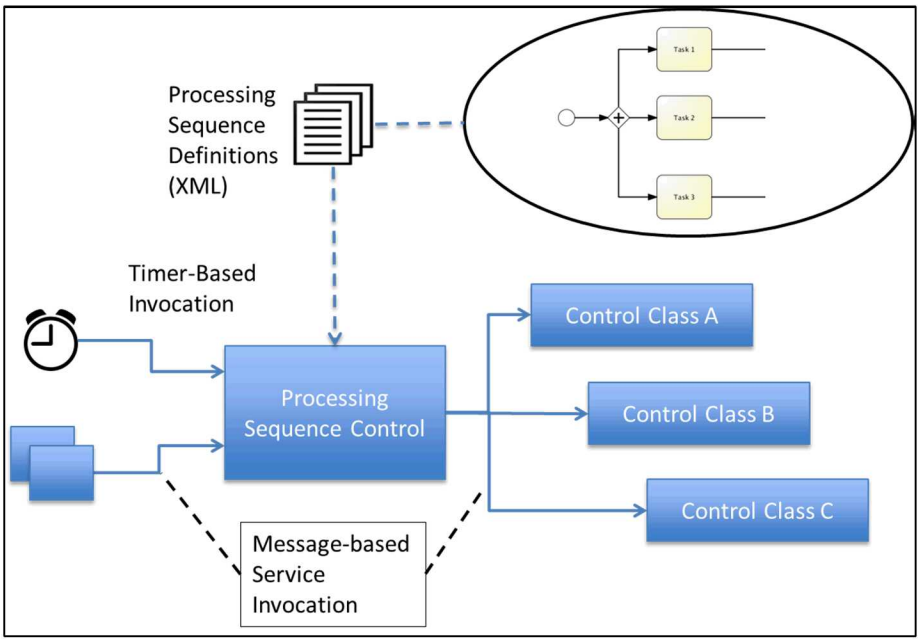

Figure 6-3. Service-Based Processing Sequence Execution

This is the last page of the document.